Securing the EDK II Image Loader

Marvin Häuser
Technische Universität Kaiserslautern;
Ivannikov Institute for System Programming
of the Russian Academy of Sciences
Kaiserslautern, Germany
mhaeuser@posteo.de

Vitaly Cheptsov
Ivannikov Institute for System Programming
of the Russian Academy of Sciences
Moscow, Russia
cheptsov@ispras.ru

***Abstract*—The Unified Extensible Firmware Interface (UEFI) is a standardised interface between the firmware and the operating system used in all x86-based platforms over the past ten years, which continues to spread to other architectures such as ARM and RISC-V. The UEFI incorporates a modular design based on images containing a driver or an application in a Common Object File Format (COFF) either as a Portable Executable (PE) or as a Terse Executable (TE). The de-facto standard generic UEFI services implementation, including the image loading functionality, is TianoCore EDK II. Its track of security issues shows numerous design and implementation flaws some of which are yet to be addressed. In this paper we outline both the requirements for a secure UEFI Image Loader and the issues of the existing implementation. As an alternative we propose a formally verified Image Loader supporting both PE and TE images with fine-grained hardening enabling a seamless integration with EDK II and subsequently with the other firmwares.**

***Index Terms*—UEFI, Parsing, Security, Program verification.**

I. Introduction

The fast spread of the Unified Extensible Firmware Interface (UEFI) [1] over x86-based platforms would not have been possible without the adoption of other specifications backed with existing widely available tools. The use of the Portable Executable (PE) [2] format let Intel reuse the existing specification for the format itself as well as the established compiler infrastructure. This decision led the UEFI Forum to keep including the PE format as part of the UEFI specification essentially making it the core of the UEFI modular architecture.

The UEFI architecture is designed in a way that not all the modules used during the boot sequence can be equally trusted. While firmware module trust is assumed as a fact by the UEFI and UEFI PI [3] specifications and is handled by vendor-specific technologies such as Intel BootGuard [4], other module protection is not taken for granted and is part of the "Secure Boot and Driver Signing" and "Secure Technologies" sections of the UEFI specification.

A driver coming with an option ROM, a UEFI application, or a standalone UEFI driver are by all means less trusted but usually do not execute with significantly reduced privileges compared to firmware drivers and applications. To circumvent this disadvantage each module external to the firmware is verified to belong to an authority that is trusted by the firmware prior to execution. The standardised verification algorithm in UEFI for image signature is Microsoft Authenticode [5], which bundles image signature within the PE file and requires most of structural parsing for its verification. While the choice of this algorithm made it possible to reuse the code signing tools also used for Windows driver signing easing the development process, it also imposed a number of extra security requirements on the software in charge of the signature verification process on the firmware side.

Today most of the platform-independent code of the UEFI firmwares is implemented as part of the open-source TianoCore EDK II [6] toolkit. This implementation consisting of about a million lines of C code also includes the code to handle the PE images that can be found in practically any modern firmware. In an attempt to make the implementation more reliable contributions to EDK II go through a thorough review process backed with additional specifications covering the development from architecture to code style, automated testing on several platforms and on-demand testing with several static and dynamic analysers [7]. The amount of vulnerabilities found in modern firmwares over the last ten years shows that the measures taken are clearly not enough [8]. While the attempts to introduce other programming languages with built-in safety checkers may improve the situation with naive security issues like buffer overflows, logic errors are much harder to find and in general cannot be identified automatically.

In this paper we suggest using formal verification to secure the PE image loading functionality in UEFI. Section II describes the issues of the current implementation present in EDK II and specifications separating design defects, functional defects, and potential issues not covered by the existing implementation. Section III provides a basic introduction into formal verification and introduces AstraVer, the tool we used to verify the code against our models. Section IV describes the implementation and the verification model.

Note: We would like to clarify on our terminology to avoid ambiguities. Please note that specific terms from the cited specifications will not be repeated.

- "[Raw] file" shall mean the format in which a PE or TE image is saved on storage devices.
- "[Loaded] image" shall mean the format of a PE or TE image expanded into the execution environment.
- "Verification" shall mean the process of ensuring the plausibility of data structures within a file or image

conforming to the PE format [2].
- "Hashing" shall mean the cryptographic hashing of the raw file data in a way described in the used signature verification algorithm such as Authenticode [5].
- "Loading" shall mean the transformation of a TE or PE raw file into a loaded image.
- "Relocating" shall mean rebasing a loaded image from one virtual address base to another.
- "Image Loader" shall mean the library that performs actions such as verification, hashing, loading or relocating of a raw file or image.

II. State of the Art

The EDK II Image Loader contains various defects. While some are bugs of the specific implementation ("functional defects"), others are fundamental design flaws of the library design that significantly raise the risk of introducing functional defects ("design defects").

A. Design defects

1) **Time-of-check/Time-of-use vulnerable design**
The EDK II Image Loader uses a function provided by the caller to abstract the gradual image data reading[1]. As made clear by the function specification its explicit purpose is to allow arbitrary, untrusted data sources such as networks. At the time of writing there are a total of 18 calls to this function across multiple stages of loading which makes ensuring data is not re-read and used unverified (*TOC/TOU*) a non-trivial task. This is currently an actual threat as the image headers are read early for verification[2] and then again when loading the image into its final memory destination[3] without ensuring equivalence or re-verifying the data read. Whether this is an actual security vulnerability depends on whether the caller-provided read function uses a trusted source (e.g. main memory) or not (e.g. network).
This defect is addressed by hardened design (IV-D).
2) **Out-of-bounds vulnerable design**
HII data that is provided to the caller[4] is variably-sized. Since its internal offsets are not validated to be in bounds of the image address space and it is also not stored with the maximum bounds, so that it can be safely accessed[5], the caller is left at a high risk of an out-of-bounds access.
This defect is addressed by hardened design (IV-D) and raw file and loaded image models (IV-C).
3) **Untrusted data in a trusted environment**
This defect closely relates to the previous defect 2, but is caller- rather than callee-centric. The UEFI specification requires the UEFI image's HII resource data, which is variable-length[6], to be exposed to the trusted environment (UEFI protocol installation). Consequently, the Image Loader locates and exposes the location of this data[7]. However, no verification occurs within or outside the Image Loader before the data is exposed at load time[8]. Loading an image cannot be seen as an act of trusting it, as with a safe loader implementation the image did not yet gain or was given any control over the system.
This defect needs to be addressed on UEFI specification level.
4) **Ambiguous context ownership**
Traditionally, context instances are owned by the library that declares them — coming from an object-orientated programming paradigm, they can be thought of as non-static members of a class instance. While public reading can be tolerated to simplify the interface, writes should be performed strictly within the library implementation to preserve consistent states. EDK II currently relies on external context writes as part of the library function contracts[9].
This defect is addressed by hardened design (IV-D).
5) **Complicated function contracts**
Function contracts should be easy to understand and control flows easy to comprehend. Presently EDK II relies on explicitly declaring the context fields required to be valid and the context fields ensured to be valid by each function[10]. Except for fields explicitly allowed to be publicly read this severely overcomplicates the documentation.
This defect is addressed by hardened design (IV-D).
6) **Decentralisation of code**
There are several locations throughout the EDK II codebase that access PE and TE image structures directly instead of utilising the Image Loader[11]. This is strongly undesired especially for the hashing algorithm which is present four times[12] outside the Image Loader, where the function logically belongs. Without the library explicitly expressing the guarantees it makes regarding image data structures (as opposed to only its own context structure) the code cannot trust the structures. Validating them then may lead to code duplication depending on the library implementation details. Furthermore, the Image Loader is exposed via two separate libraries[13] for no obvious reason. This decentralisation adds cost to review, validation and bug fixing.
This defect is addressed by hardened design (IV-D).

[1] EDK II [6] PeCoffLib.h:33-70
[2] EDK II [6] BasePeCoffLib/BasePeCoff.c:81-86
EDK II [6] BasePeCoffLib/BasePeCoff.c:112-117
[3] EDK II [6] BasePeCoffLib/BasePeCoff.c:1266-1271
[4] EDK II [6] Core/Dxe/Image/Image.c:1406-1416
[5] EDK II [6] BasePeCoffLib/BasePeCoff.c:1613
[6] EDK II [6] UefiInternalFormRepresentation.h:48-51
[7] EDK II [6] BasePeCoffLib/BasePeCoff.c:1613
[8] EDK II [6] Core/Dxe/Image/Image.c:1406-1416
[9] EDK II [6] PeCoffLib.h:208-209
[10] EDK II [6] PeCoffLib.h:199-201, EDK II [6] PeCoffLib.h:208-209
[11] EDK II [6] Core/Dxe/Mem/MemoryProfileRecord.c:251-353
[12] EDK II [6] DxeImageVerificationLib/DxeImageVerificationLib.c:273-293
EDK II [6] SecureBootConfigDxe/SecureBootConfigImpl.c:1809-1825
EDK II [6] DxeTpmMeasureBootLib/DxeTpmMeasureBootLib.c:267-300
EDK II [6] Tcg2Dxe/MeasureBootPeCoff.c:76-101
[13] EDK II [6] PeCoffGetEntryPointLib.h

7) **Centralisation of hashing ciphers**
The hashing function mentioned in "Decentralisation of code" statically defines a list of supported hashing ciphers[14]. This means that not only custom cryptography is not easily supported, there also is no easy platform control over the allowed ciphers.
This defect is addressed by hardened design (IV-D).

8) **Runtime relocation is status-less**
Unlike all other APIs, the EDK II Image Loader does not report a status for the Runtime relocation operation[15]. However, there are obvious and known error conditions where the operation may not succeed and they are handled inappropriately by silently returning or ASSERTing[16].
This defect is addressed by hardened design (IV-D).

9) **Data structures are not protected from relocation**
Base Relocation targets may point to parts of the image data structures, the most notable of which is the Relocation Directory itself. This may corrupt the relocation data making it unusable for further processes such as Runtime relocation.
This defect is addressed by the loaded image model (IV-C).

10) **Unclear relocation semantics**
The descriptions for the *IMAGE_REL_BASED LOW* and *HIGH* Base Relocation types clearly outline they are compound. However, their semantics does not define how the carry for the low component is to be handled. This already has caused confusion in the past leading to a patch that reasonably adds the carry to the high component but that also makes the assumption that those Base Relocation types are always consecutive and ordered[17], which is not guaranteed by the PE format.
This defect is addressed by hardened design (IV-D). It additionally needs to be addressed on PE format [2] level.

11) **Image information is leaked**
The image headers, debug information, and Relocation Directory contain information about exact locations such as function addresses in the image data. This information can be used to more easily locate gadgets and potentially transparently spoof certain operations when write access is gained to the image data.
This defect is addressed by hardened design (IV-D).

12) **Nondeterminism**
It is not required for Base Relocations to be processed in a specific order or to target distinct memory by the PE format. Furthermore, there are no constraints to the first image section (such as that it must be the start of the image), whether the image headers must be loaded or what needs to happen with the possible gap from their end to the beginning of the first section's memory. Hence, valid loaders may produce different results for valid images (nondeterminism).
This defect is addressed by the loaded image model (IV-C) and ACSL models[18]. It additionally needs to be addressed on PE format [2] level.

B. Functional defects

1) **Out-of-bounds accesses**
The current *PeCoffLoaderImageAddress* function returns a pointer to a requested offset within the image buffer. However, it does not return the remaining number of bytes to the end of the buffer[19] — all callers to this function that access more than one byte from this pointer without additional caution may perform out-of-bounds accesses (*OOB*). While there is a practice of calling the function twice, where the second call is passed the range's end offset[20], this is very much unintuitive and error-prone. There are actual occurrences of *PeCoffLoaderImageAddress*-based OOB accesses[21]. Furthermore, HII section lookup may access elements without ensuring their prior existence[22]. It is also not explicitly documented that requests for data in the TE header are not supported due to the *TeStrippedOffset* subtraction.
This defect is addressed by the raw file and loaded image models (IV-C).

2) **Integer wraparounds**
The EDK II Image Loader is affected by multiple types of integer wraparounds. Some of them are likely harmless[23] because they are implicitly accounted for shortly after[24], yet this cannot be allowed as it makes manual review much harder and is error-prone regarding future refactoring. There also are some that may cause an infinite loop upon facing unlucky values[25].
This defect is addressed by the raw file and loaded image models (IV-C).

3) **Alignment requirement violations**
Any CPU architecture may impose alignment requirements for data access. Prominent examples include x86 (SSE), ARM, MIPS and PowerPC, of which the first two are officially supported by the UEFI right now. While most unsupported unaligned accesses result in exceptions, some, e.g. ARMv6 and below[26], may yield unpredictable behaviour[27]. The EDK II Image Loader does not verify alignment requirements of offset-based pointers[28]. In fact EDK II does not provide any way to do so at the time

[14]EDK II [6] DxeImageVerificationLib/DxeImageVerificationLib.c:322-342
[15]EDK II [6] PeCoffLib.h:356
[16]EDK II [6] BasePeCoffLib/BasePeCoff.c:1748-1760
[17]www.mail-archive.com/edk2-devel@lists.sourceforge.net/msg16005.html
[18]They are exclusive to the code and are not discussed in this document.
[19]EDK II [6] BasePeCoffLib/BasePeCoff.c:843-848
[20]EDK II [6] BasePeCoffLib/BasePeCoff.c:1743-1746
[21]EDK II [6] BasePeCoffLib/BasePeCoff.c:1538-1542
[22]EDK II [6] BasePeCoffLib/BasePeCoff.c:1583
EDK II [6] BasePeCoffLib/BasePeCoff.c:1600
[23]EDK II [6] BasePeCoffLib/BasePeCoff.c:139
[24]EDK II [6] BasePeCoffLib/BasePeCoff.c:162-174
[25]EDK II [6] BasePeCoffLib/BasePeCoff.c:701 (for 32-bit $UINTN$)
[26]Supported as per UEFI specification [1], 2.3.5 AArch32 Platforms
[27]ARMv6-M [9], A3.5.5 Memory access restrictions
[28]EDK II [6] BasePeCoffLib/BasePeCoff.c:1273

of writing by neither allowing usage of the *_Alignof* operator that is part of the C Programming Language [10] nor providing a macro of its own.
Furthermore, there are occurrences of aligned access to data that may legitimately be unaligned[29].
This defect is addressed by hardened design (IV-D) and the raw file and loaded image models (IV-C).

4) **Uninitialized destination bytes**
The EDK II Image Loader does not initialize the destination buffer sufficiently[30]. This means that the destination area may contain arbitrary bytes that are not covered by the image signature. Bugs in the image code, the Image Loader, or similar spots may lead to an unexpected attack vector. As the problem areas are part of the image memory, tools will most likely have trouble detecting this.
This defect is addressed by the loaded image model (IV-C).
5) **Function specification violation**
The Image Loader currently reports success unconditionally when image relocation is requested but relocation information has been stripped. Especially it is not verified whether the destination address matches the preferred image address[31]. This violates the function specification which states the image has been relocated when success is returned[32].
This defect is addressed by ACSL models.
6) **Runtime relocation is optimistic**
The PE format has no concept to support the relocation processing UEFI needs to perform when entering OS Runtime[33]. Yet UEFI requires images to be relocated after they have already been executed. This may change values at offsets targeted by Base Relocations. The EDK II Image Loader solves this issue by optimistic bookkeeping as to changed values are skipped[34]. However, the PE format does not consider loader behaviour of this kind and thus must be restricted to not accidentally compromise security.
This defect is addressed by ACSL models.
7) **HII section lookup may malfunction**
The current implementation of the HII section lookup may cause unexpected behaviour by unintentionally exchanging the loop object ("ResourceDirectory") during the loop execution[35].
This defect is addressed by ACSL models.
8) **Sections might overlap**
In theory sections may refer to overlapping bytes. This would not be a problem with an algorithm hashing the entire binary at once but the Microsoft Authenticode algorithm hashes every section individually, which means bytes may be hashed multiple times. This fact can be abused to effectively hash a dramatically higher amount of bytes in total than if overlapping sections were prohibited. While no evidence of such an attack vector seems to be available at this point, it cannot be taken out of consideration as a viable threat in the future.
This defect is addressed by hardened design (IV-D) and ACSL models.
9) **TE sections or header may be loaded unaligned**
For TE images the Image Loader locates virtual addresses by subtracting the TE stripped size first[36]. Because those fields remain untouched by the PE to TE conversion, this results in an address not aligned by the image section alignment. This behaviour is not documented by the Image Loader itself, however, the *PEI Core* works around this by adding the TE stripped size to the destination address[37]. This in return results in the TE image header not being loaded at an address aligned by the image section alignment. The *DXE Core* does not attempt to work around this at all.
This is especially bad for eXecute In Place (XIP) images as they are not loaded at all and thus remain unaligned in the flash memory. However, XIP images are out of the scope of this document.
This defect is addressed by the loaded image model (IV-C).
10) **Non-conformant MS-DOS Stub is tolerated for TE**
According to the TE specification *StrippedSize* bytes are stripped from the start of the file before the TE header is added[38]. This implies there must not be any additional headers preceding the TE header. EDK II however supports a preceding MS-DOS Stub for TE images in the Image Loader[39].
This defect is addressed by the loaded image model (IV-C).
11) **TE *SizeOfHeaders* is inadequate**
EDK II defines *SizeOfHeaders* for TE images via *BaseOfCode*[40] for no obvious reason. Unfortunately *SizeOfHeaders* cannot be reconstructed precisely from the TE image, however *BaseOfCode* may cause non-obvious issues with images that have an unexpected section order.
This defect is addressed by ACSL models. It additionally needs to be addressed on PI specification [3] level.

C. Unaddressed considerations

1) **Base Relocations might overlap**
Theoretically Base Relocations may refer to overlapping bytes. While this obviously is unreasonable, the PE format [2] does not cover such a possibility or how to avoid it. As there is no guarantee of their order or similar,

[29] EDK II [6] BasePeCoffLib/BasePeCoff.c:1060
[30] bugzilla.tianocore.org/show_bug.cgi?id=1999
[31] EDK II [6] BasePeCoffLib/BasePeCoff.c:923-931
[32] EDK II [6] PeCoffLib.h:248
[33] UEFI specification [1], 8.4 Virtual Memory Services
[34] EDK II [6] BasePeCoffLib/BasePeCoff.c:1815
[35] EDK II [6] BasePeCoffLib/BasePeCoff.c:1549
[36] EDK II [6] BasePeCoffLib/BasePeCoff.c:858
[37] EDK II [6] Core/Pei/Image/Image.c:384-395
[38] PI specification [3], Volume 1: PEI Core, 15.2 PE32 Headers, TE Header
[39] EDK II [6] BasePeCoffLib/BasePeCoff.c:79-129
[40] EDK II [6] BasePeCoffLib/BasePeCoff.c:139

expensive bookkeeping would be required to ensure all Base Relocations refer to disjoint ranges. Accounting for this does not seem to be beneficial as all the operations are memory-safe and overall safety cannot be guaranteed in either case.

2) **Overcomplicated hashing algorithm**
The PE hashing algorithm specified by the Authenticode format[41] has been designed for a debugging-friendly in-OS usage, mostly to allow file modifications after the signing process. However, especially in security-critical low-level software, this is not permitted. Instead, it would be more advisable to use a simpler hashing algorithm that hashes the file entirely (manifest-based verification) or till the trailing signature. The PE format guarantees the certificate and signature information are indeed trailing[42], so the most intuitive way to hash the file is to hash all bytes from the beginning of the file to the start of this data — the related security directory information must be set before the image is hashed.
3) **Hashing as early as possible**
Please refer to "Overcomplicated hashing algorithm" (2) for context. It is common practice to hash a binary and validate its signature as early as possible in the loading process to avoid abusing loader bugs. However, since PE has a very complex hashing algorithm many of the image properties need to be verified beforehand. Therefore, it is unreasonable to postpone the rest of the validation.
4) **Image fields not covered by the hash**
The Authenticode algorithm for PE image hashing dictates skipping the checksum and the security directory information of the image header[43]. Good security practice suggests that these fields shall be zero in the destination area. Meanwhile, alternative algorithms have surfaced, such as in Mac EFI where hashing operates on the entire file till the signature information. In this case those fields must be loaded into the destination area. Considering the usage of an alternative hashing algorithm is strongly recommended, zeroing the three affected fields is likely not worth it. Furthermore, the new design allows omitting loading the headers explicitly and at the same time allows them to be covered by a section, both of which do not require such actions.

III. AstraVer Toolset

Formal verification is one of the software verification techniques commonly utilised in safety-critical software development lifecycles in aerospace, automotive, medical, energetic, and other similarly high-risk industries. The aim of formal verification is to mathematically prove the correspondence of an algorithm to its formal specification in order to be able to reason about the safety and correctness of the implementation and reducing the impact of a human error.

[41] Authenticode [5], Calculating the PE Image Hash
[42] PE format [2], The Attribute Certificate Table (Image Only)
[43] Authenticode [5], Calculating the PE Image Hash

Although formal verification is generally considered heavy-weight and time-consuming, modern tools involving various verification strategies with automated reasoning make it possible to verify real-world software. Among the well-known success stories are avionics software verification in Airbus [11] done primarily with Caveat and Frama-C, Astra Linux security module verification with the AstraVer plugin for Frama-C [12], Hyper-V hypervisor verification in Microsoft with VCC [13], seL4 microkernel in OK Labs with Isabelle/HOL [14], and CompCert C compiler at INRIA with Coq [15].

In general, there are two formal verification approaches:

- Synthesising program code from the formal proof.
- Verifying already written program code.

The choice between the two strongly depends on the nature of the target software and the ability to modify it. Synthesising a formally verified program commonly allows proving larger software or software with complex mathematical algorithms such as High Assurance Cryptographic Library (HACL*) [16]. On the other side verifying an existing program allows for uninterrupted development by separating the deadlines of a working prototype and fully verified software in the lifecycle and enables the ability to integrate legacy and third-party software without the need to rewrite it from the ground up. When verifying large codebases, the general approach is to only write formal proofs for most critical areas and use other means of software verification for the other making the process more affordable.

Although no test suite can provide the guarantees comparable to formal verification simply because the number of tests is always finite, there still are various methods that have proven to be effective in safety and security areas, like tests with full MC/DC coverage or fuzzing. These methods should not be neglected and may be used together with formal verification. It should always be taken into account that while we significantly improve software reliability by providing a formal proof, mistakes can still happen within the proof, the proving tool itself, or on other layers like in the compiler or even in the hardware.

Since the aim of this project was to demonstrate the ability to create formal proofs for existing software in the UEFI environment, we chose the AstraVer Toolset. This tool allows expressing function behaviour as a specification written in a dedicated language called ACSL [17] consisting of a precondition and a postcondition. The tool translates annotated C code into a set of logical formulae (verification conditions), the general validity of which is equivalent to program correctness as in Floyd-Hoare logic.

The choice of the AstraVer Toolset over the base Frama-C version was dictated by the wider set of abilities required for proving real-world software with pointer operations, bitwise arithmetic, type inspection, and the like. In this sense AstraVer Toolset is a further development of the Jessie plugin [18] for Frama-C tested on existing system software such as the Linux kernel. It implements a new memory model [19] that allows to support the *container_of* construct, pointer type reinterpretation between integer types, including types of different

size, bitwise arithmetic operations on expression-level, and has several other features [12].

IV. Implementing the Image Loader

A. Memory model

We assume the CompCert Memory Model v2 [20] as a base for the proof design. Punctually its guarantees are relaxed towards the previous CompCert Memory Model v1 as it is closer to the model AstraVer uses internally[44]. Namely:

- For functions performing only read operations valid pointers of different types are not required to refer to disjoint memory regions from valid byte arrays.
- For functions performing direct write operations valid pointers of different types must all refer to disjoint memory. Most notably no memory may overlap between a byte array and a different data type. This must either propagate to the caller, or the *assigns* clause must include all representations (e.g. the full byte range covered by a modified data structure), or there must be some obvious internal proof the data did not change.

The reason for this is that to AstraVer valid pointers of different types always point to disjoint memory. For reading operations this is acceptable. If a write action is performed to a valid pointer however, the change will not be reflected for any other valid pointer of a different type partially or fully aliasing the same memory rendering the proof inaccurate.

For the production environment, we fully assume the v2 model to introduce performance improvements such as reading aligned 32-bit values in one operation. To accomplish this, we use macros that in the proof environment operate on byte-level but still have correct alignment constraints.

B. Type model

1) Data type alignment model: According to the UEFI specification, all fundamental types are aligned naturally. For structures it declares they are aligned by the maximum size out of all internal data[45], but real-world compilers use the maximum internal alignment instead. Natural alignment in this context refers to alignment being equal to the minimum of the data type size and the maximum alignment size which is defined by the architecture's UEFI ABI. Please remember that floating-point types are not supported by UEFI.

These assumptions were used to formulate an acceptable alignment model in the lack of a tool-supported one:

$$A\_MAX := \begin{cases} 4 & \text{for IA32} \\ 8 & \text{otherwise} \end{cases}$$

$$\_Alignof(T) \stackrel{\text{def}}{=} \min\{sizeof(T), A\_MAX\}$$

For fundamental types and the UEFI ABI this definition is accurate. For aggregate types and unions however, the alignment our type model yields is often too high as the type size is often larger than the maximum alignment of each internal datum. Such alignment proofs are considered "proof of concept" rather than an actual part of the formal verification, and we must admit our current toolset does not allow for an accurate result. A new revision of the AstraVer Toolset is currently in development and the mentioned culprits are considered for its design.

We addressed this limitation as such for the moment. When an operation or an ACSL annotation lead to a pointer known to be aligned for one type also be aligned for the other, a $\_Static\_assert$ is used to prove transitivity in case not all distinct types are explicitly verified.

Considering the data type definitions and their sizes, data type alignments can only ever be a power of two. Powers of two especially satisfy the following condition of the C Programming Language standard, which we used in multiple spots for code optimisations: "When an alignment is larger than another it represents a stricter alignment."[46]

2) Pointer target alignment model: The pointer target alignment model is incomplete but sufficient for all required use cases. Any memory allocation function declares a guarantee for the alignment of the returned pointer (e.g. 4 KB alignment for page-wise allocation or 8-byte alignment (A_MAX) for $AllocatePool$). From there, additions to these pointers are axiomatically modelled to yield a pointer of alignment a if the pointer (p) and offset (o) are both aligned by a.

$$\forall p, a, o : aligned(p, a) \land o \bmod a = 0 \Rightarrow aligned(p + o, a)$$

This is correct as per the data type alignment model as the sum of two values divisible by a power of two itself is also divisible by that power of two.

3) Pointer target validity model: The C Programming Language does not have a standard-compliant way to identify the type of data structures from the in-memory representation. While we cannot give a definition for the sufficient conditions to determine validity, we can define it based on the required ones of both the C Programming Language standard and the PE format:

For all the file and image locations the PE format designates a data structure for, it is valid if and only if the entirety of its size is contained in the bounds of the file or image and the location satisfies the alignment requirements of the designated data type.

4) Fundamental type definitions: In the following we define the utilised fundamental data types of our Image Loader derived from the UEFI specification[47].

From the generic definitions

$$FALSE := 0, TRUE := 1$$

$$UINT(x) := \{n \in \mathbb{N} \mid 0 \leq n < 2^x\}$$

[44] Linux kernel verification [12], 4 Region separation in Jessie, 5.1 Jessie byte-level block memory model

[45] UEFI [1], 2.3.1 Data Types

[46] C17 [10], 6.2.8.7 (unchanged since C11)

[47] UEFI [1], 2.3.1 Data Types

we define the generic fundamental types. Please note that for simplicity, we assume that $CHAR8$ is unsigned.

Type	Definition	sizeof	_Alignof
$BOOLEAN$	$\{TRUE, FALSE\}$	1	1
$CHAR8$	$UINT(8)$	1	1
$UINT8$	$UINT(8)$	1	1
$UINT16$	$UINT(16)$	2	2
$UINT32$	$UINT(32)$	4	4
$UINT64$	$UINT(64)$	8	A_MAX

$$UINTN := \begin{cases} UINT32 & \text{for IA32, ARM} \\ UINT64 & \text{for X64, AArch64, RISC-V} \end{cases}$$

5) Aggregate types and unions: The composition of fundamental data types and recursively aggregate types and unions follows the C Programming Language. Some constructs cannot be expressed in the C Programming Language, such as arrays of variably-sized data — other means of modelling such as regular expressions will be used in the following to express their composition. Data structures modelled in the referenced specifications, such as the UEFI or the PE specifications, that are expressible in the C Programming Language are assumed to be known and will not be explicitly modelled.

C. Proof definition

1) Model-aided proof goals: To increase the security and stability of the Image Loader, we specified the following as our verification requirements. In particular, they include considerations in context of the C Programming Language.

Safety requirements:

- Non-modulo integer arithmetic does not wrap around.
- All functions terminate on all external inputs.
- Memory accesses happen in valid bounds and are aligned.

Functional requirements:

- Raw files that do not conform to the modelled subset of the PE or TE file formats are discarded.
- The image is loaded correctly and the result is deterministic (please refer to design defect 12).
- Data expected to remain valid is not invalidated (e.g. applying Base Relocations does not modify the Relocation Directory itself, please refer to design defect 9).
- Individual Base Relocations are applied correctly.

The safety of integer operations and function termination are automatically proved by AstraVer unless specified otherwise. No such conditions have been defined for our Image Loader, hence both goals are always met.

To satisfy the remaining requirements, we will define predicates based on the PE and TE specifications, as well as our own constraints that compose the file and image models ("format model"). They are translated to ACSL definitions and the code is annotated in such a way that non-conformant inputs abort the process.

2) Notation: Below you will find a table of defined operators, functions, acronyms and abstractions.

#	Amount of elements in a collection
alignBRT(r)	Base Relocation target alignment
align(v,a)	The least multiple of a not less than v
size(d)	Size of the variably-sized data structure
sizeBRT(r)	Base Relocation target size
vaBRT(r)	Base Relocation target VA
typeBRT(r)	Base Relocation target data type
BR	Base Relocation (inner-page)
BRB	Base Relocation Block (page-wise)
BS	Block size
DOS	MS-DOS Stub
H	Raw file headers
O	Offset
$PE32$	COFF and PE32 Optional Header
$PE32Plus$	COFF and PE32+ Optional Header
RD	Relocation directory
RS	Size of raw file data
S	Section Table ($S = (SH)^+$)[48]
SH	Section header
T	Type
TE	TE Header
VA	Virtual address
VS	Virtual size
$c.FS$	Raw file size
$c.HS$	Raw file *SizeOfHeaders*
$c.IS$	Image *SizeOfImage*
$c.RDS$	Image Relocation Directory size
$c.RDV$	Image Relocation Directory VA
$c.SA$	Image *SectionAlignment*
$c.SO$	Raw file Section Table offset

The operators not mentioned in the table are defined as in the C Programming Language standard. Please remember that $sizeof$ does not include the size of flexible arrays. c is used in the following to denote an image context, i.e. image loader state structure, to reason in an image type agnostic fashion.

3) File headers: As previously described, the memory model allows for overlapping regions of different data types for read operations. The raw file is read-only throughout the library, thus no considerations regarding overlapping need to be made. We require all file buffers to be aligned by the maximum fundamental alignment (A_MAX). Please note that TE stripping will not be considered for simplicity.

$$PE = PE32|PE32Plus$$
$$H = (TE|[DOS] \circ {}^{49}PE)^{50}$$

There are several constraints, especially for size fields. However, they will not be modelled for being obvious from

[48] The requirement of at least one section is imposed by us.

[49] The $\circ$ operator denotes concatenation.

[50] PE32, PE32Plus, and DOS data structures may be succeeded by arbitrary data or padding which is considered to be part of its definition for simplicity. This trailing data is sized to hold the alignment for the following data.

the specifications. Due to the flexible format of the headers, formal definitions will be omitted, but the following must hold:

- The PE header offset from the MS-DOS Stub must be aligned.
- $size(H) \leq c.FS$

The initialization routine verifies the input header and succeeds if and only if it matches the definition of H. This allows us to prove the following:

- The file headers are all in bounds and aligned.
- The file headers conform to the format model.

4) File Section Table: The image's virtual address space is composed of one or more sections which must be sorted in ascending order and be contiguous in terms of section alignment[51]. It is obvious that the loaded sections are disjunct. Their headers reside adjacent to H (thus we have $H \circ S$). For PE32 and PE32+ images they must also fit their *SizeOfImage* value.

While the PE format does not explicitly describe the constraints of the first virtual address, most tools set it to the aligned end address of the image headers because they expect the Image Loader to manually load it into the execution environment. However, the headers should not be accessed except by the Image Loader and debugging instruments in UEFI. Thus, we decided to make explicitly loading them optional, to prohibit section file data to overlap with them and to allow the first section to be the beginning of the address space. Both allowed values for the first Virtual Address are aligned by *SectionAlignment* and hence all sections are correctly aligned[52].

$$
\begin{aligned}
&correctSA(c, s) \stackrel{\text{def}}{=} \\
&\#s > 0 \wedge (s[0].VA = 0 \vee s[0].VA = align(c.HS, c.SA)) \wedge \\
&\bigwedge_{i=1}^{\#s-1} s[i].VA = align(s[i-1].VA + s[i-1].VS, c.SA) \wedge \\
&align(s[\#s-1].VA + s[\#s-1].VS, c.SA) \leq c.IS
\end{aligned}
$$

Every section has an offset into the file buffer at which its data is located as well as its size — obviously, the file bounds must be respected. Please note that the size that will be copied by the loading code for every section header sh is $\min sh.VS, sh.RS$ and $[sh.VA + sh.RS, sh.VA + sh.VS)$ is filled with zeros.

$$
\begin{aligned}
&validMemS(c, s) \stackrel{\text{def}}{=} \\
&c.SO \bmod \_Alignof(s[0]) = 0^{53} \quad \wedge \\
&c.SO + \#s \cdot sizeof(s[0]) \leq c.FS \quad \wedge \\
&\bigwedge_{sh \in s} 0 < sh.RS \Rightarrow c.HS \leq sh.O \wedge sh.O + sh.RS \leq c.FS
\end{aligned}
$$

[51] PE Format [2] "Section Table (Section Headers)"

[52] Malformed binaries reached production due to the current Image Loader failing to verify those properties. In response, we introduced an optional mode which does not verify strict continuity or alignment but only the derived properties such as ascending order and separation of section memory. Its model is out of scope.

The constraints regarding file and image memory add up to the section correctness.

$$correctS(c, st) \stackrel{\text{def}}{=} correctSA(c, st) \wedge validMemS(c, st)$$

The initialization routine verifies the input Section Table and succeeds if and only if they satisfy $correctS$. This allows us to prove the following:

- The section headers are all in bounds and aligned.
- The section headers conform to the format model.
- The raw and virtual section targets are all in bounds (thus loading does not cause OOB).
- The virtual address space is contiguous and thus deterministic.
- Loading from the file buffer to the image memory is injective (thus loading a section does not invalidate data loaded by previous sections).

5) Image Relocation Directory: The Relocation Directory is a concatenation of arbitrarily many Base Relocation Blocks.

$$RD = (BRB)^*$$

We define $size()$ functions to be able to clearly express their bounds.

$$
\begin{aligned}
size(brb) &\stackrel{\text{def}}{=} sizeof(BRB) + \#brb.BR \cdot sizeof(brb.BR[0]) \\
size(rd) &\stackrel{\text{def}}{=} \sum_{brb \in rd.BRB} size(brb)
\end{aligned}
$$

Base Relocation targets vary in target data type, data size and data alignment requirements per type. For the scope of this project, we only allow a subset of the Base Relocation types found in the real world. They are characterized as such (where $C8$ and $U32$ denote $CHAR8$ and $UINT32$):

$$
typeBRT(br) \stackrel{\text{def}}{=} \begin{cases} C8[4] & \text{if } br.T = HIGHLOW \\ C8[8] & \text{if } br.T = DIR64 \\ U32[2] & \text{if } br.T = ARM\_MOV32T \\ VOID & otherwise \end{cases}
$$

$$
alignBRT(br) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } br.T = HIGHLOW \\ 1 & \text{if } br.T = DIR64 \\ 4 & \text{if } br.T = ARM\_MOV32T^{54} \\ \infty & otherwise \end{cases}
$$

$$
sizeBRT(br) \stackrel{\text{def}}{=} \begin{cases} 4 & \text{if } br.T = HIGHLOW \\ 8 & \text{if } br.T = DIR64 \\ 8 & \text{if } br.T = ARM\_MOV32T \\ \infty & otherwise \end{cases}
$$

$$vaBRT(brb, br) \stackrel{\text{def}}{=} brb.VA + br.O$$

For the target of a Base Relocation, its start must be correctly aligned for its type and its range must not overlap with the Relocation Directory memory. This particularly satisfies the memory model condition of aliased data (the byte and the data

[53] This constraint is our own and follows from the type model.

[54] This requirement is not specified by the PE format, but is derived from the ARM architectural requirements.

structure representations of the Relocation Directory) being constant.

$$t := vaBRT(brb, br), s := sizeBRT(br)$$
$$correctRT(c, brb, br) \stackrel{\text{def}}{=}$$
$$t + s \leq c.IS \wedge t \bmod alignBRT(br) = 0 \quad \wedge$$
$$[t, t+s) \cap [c.RDV, c.RDV + c.RDS) = \emptyset$$

With this, we can define the remaining correctness.

$$correctBR(c, brb, br) \stackrel{\text{def}}{=}$$
$$sizeBRT(r) < \infty \wedge correctRT(c, brb, br)$$
$$correctBRB(c, brb) \stackrel{\text{def}}{=}$$
$$brb.BS \bmod \_Alignof(BRB)^{55} = 0^{56} \quad \wedge$$
$$brb.BS = size(brb) \wedge \forall r \in brb.BR : correctBR(c, brb, br)$$
$$correctRD(c, rd) \stackrel{\text{def}}{=}$$
$$c.RDV \bmod \_Alignof(BRB) = 0 \quad \wedge$$
$$c.RDV + c.RDS \leq c.IS \wedge c.RDS = size(rd) \quad \wedge$$
$$\forall brb \in rd.BRB : correctBRB(c, brb)$$

The relocation routine verifies the input Relocation Directory and succeeds if and only if it satisfies $correctRD$. We require Base Relocations to be processed in the order of their appearance. This allows us to prove the following:

- The Relocation Directory and all its targets are in bounds and aligned.
- The Relocation Directory conforms to the format model.
- Relocation does not invalidate data used by the loader (most notably, applying Base Relocations does not modify the Relocation Directory itself).
- The resulting memory is deterministic for each distinct load address.

A proof of the correct application of individual $HIGHLOW$ and $DIR64$ relocations is available. However, due to the lack of a guarantee that the targets do not overlap, modelling the result would have been too expensive. Application effects beyond invalidation are out of scope for this document.

6) Model-aided proof results: The format model allows us to prove all previously defined goals as can be seen in sections IV-C3 to IV-C5. However, please note the following:

- The image model only mostly covers the image data structures and does not cover algorithms at all. Please refer to the Image Loader codebase.
- Only data that is used by the Image Loader is modelled and validated. Any uninvolved data may be invalid.
- The defined alignment model is not sufficient to conform to the C Programming Language (for aggregate types and unions). Additional means of ensuring correctness were taken such as manual code review, introduction of $\_Static\_assert$ usages and dynamic testing.
- While all core code was proven for safety, optional code such as ARM or RISC-V Base Relocations, the Runtime relocation bookkeeping buffer and the hashing algorithm were not proven for correct functionality. Instead, all the optional code was manually reviewed for correctness due to the proving efforts required.
- Code beyond the scope of the core Image Loader such as explicit debug directory loading and HII resource section lookup have not been proven at all but went through manual code review and dynamic testing. They are entirely optional and we expect them to be disabled in environments with special security requirements.

[55] The PE format explicitly requires 32-bit alignment which however is equivalent to $\_Alignof(BRB)$ on all platforms.

[56] This imposes a constraint on $\#brb.BR$. The $ABSOLUTE$ Base Relocation type is used to pad Base Relocation Blocks to an aligned size.

D. Hardened design

Beyond proving correctness, several changes have been made to the code design to allow for a more stable and secure operation that satisfies security-critical demands.

- Only main system memory is allowed as a data source for raw files (addresses design defect 1).
- Authenticode hashing is integrated into the core library code (addresses design defect 6).
- The hash function works similarly to *HashUpdate* (addresses design defect 7).
- Loading the header and debug information is optional (partially addresses design defect 11).
- A section discarding API is provided (partially addresses design defect 11).
- Uncommon ambiguous Base Relocation types are not supported (addresses design defect 10).
- All public functions that deal with variably-sized structures take size as a parameter or output its maximum or exact size (addresses design defect 2).
- Public context reads or writes are not part of the library design (addresses design defect 5 and partially addresses design defect 4).
- Public functions are made available for all tasks a caller needs to perform (partially addresses design defect 4).
- All public functions return a result that can indicate errors (addresses design defect 8).
- Authenticode can optionally refuse to hash overlapping raw section data (addresses functional defect 8).
- ARM and RISC-V Base Relocations are optional independently of the target architecture allowing for more platform flexibility.

V. Conclusion

Creating secure software is a problem continually addressed by improving software development life-cycle processes and involving various software verification techniques. Upon the exploration of the EDK II codebase, which is a de-facto standard implementation base for the majority of the modern UEFI firmwares, we were able to identify numerous defects accompanied by an undesirable track of security issues found in the past. With the entire UEFI modular architecture being

at a risk due to a potential compromise of the EDK II image loader we created our own implementation that can be used as a drop-in replacement upstream with only few changes required.

In order to avoid the issues of the original implementation, we thoroughly analysed the code and provided a detailed report of the found defects. This review showed that while some issues were essentially programmer errors that can be resolved by submitting patches, such as missing bounds checking or memory initialization, several others are design defects, like context ownership violations or TOC/TOU issues with untrusted storage or networking. The state-of-the-art makes creating a new Image Loader with a similar interface more practical for verification and maintenance than trying to update the existing one.

During the development of the new UEFI Image Loader we applied industry-standard practices for security-critical software in both the design and the toolset. We separated the optional code generally unintended for use in production environment, reworked the ownership semantics, enabled the use of bounds checking arithmetic, and addressed all the other defects identified during the analysis stage. Some issues found in the implementation caused by the ambiguous parts of the UEFI and PE specifications were also addressed based on real-world compiler implementations and existing images. Both core and optional code underwent static analysis with SVACE and continual fuzzing with libFuzzer reaching full line and branch coverage, except for the defensive code.

In order to provide higher reliability guarantees for the core code we created a formal proof with the help of industry-standard software used for formal verification in safety-critical systems: Frama-C with the AstraVer plugin. With accordance to the used verification model the image is discarded when it does not comply to the PE or TE specification, and is correctly loaded when it does. We proved that the loaded image can also be correctly relocated. In addition, we proved alignment, arithmetic, memory safety, and function termination. For the proof we used CVC3, CVC4, and Z3 solvers. To reduce the possibility of a true negative bug in the solver or its integration, as it happened for us several times with Alt-Ergo, we ensured that most of the verification conditions are proved by at least two solvers.

Since the UEFI environment does not have hard real-time requirements and does not specify stack capabilities, the formal proof for worst-case execution time (WCET) and stack usage are outside the scope of this paper. However, the recursion-free design with no variable-length arrays lets us be confident of a possibility of this happening in the future. For the time being the UEFI watchdog services are an acceptable countermeasure for this class of software.

We tested the implementation on a number of existing UEFI applications and DXE drivers including Apple macOS, GNU Linux, and Microsoft Windows bootloaders and supplemental drivers in the firmwares from 2013 to date with no issues discovered. Considering the results, we hope that our experience encourages the UEFI industry to incorporate this piece of software in their firmwares and use our approach to provide formal proofs for other critical firmware components.

Acknowledgements

We would like to thank the members of the ISP RAS Linux Verification Center, Mikhail Mandrykin and Alexey Khoroshilov in particular, for invaluable help and feedback on the AstraVer Toolset and general approach. We are also grateful to all our colleagues and the Ivannikov ISP RAS Open committee for their reviews, especially Laszlo Ersek from Red Hat for incredible patience and a most thorough analysis.

References

[1] UEFI Forum (2020) Unified Extensible Firmware Interface (UEFI) Specification, Version 2.8 Errata B.
[2] Microsoft Corporation (2020) PE format.
[3] UEFI Forum (2019) Platform Initialization (PI) Specification (Version 1.7).
[4] Intel Corporation (2013) Intel Hardware-based Security Technologies for Intelligent Retail Devices.
[5] Microsoft Corporation (2008) Microsoft Windows Authenticode Portable Executable Signature Format (Version 1.0).
[6] TianoCore (2020) EFI Development Kit II, commit 6c8dd15c4ae4 “SecurityPkg: Add RPMC Index to the RpmcLib”.
[7] Brian Richardson, Chris Wu, Jiewen Yao, Vincent J. Zimmer (2019) Using Host-based Firmware Analysis to Improve Platform Resiliency.
[8] Alex Matrosov, Eugene Rodionov (2017) UEFI Firmware Rootkits: Myths and Reality. Black Hast Asia 2017.
[9] ARM Limited (2010) ARMv6-M Architecture Reference Manual, Version C.
[10] ISO/IEC/JTC (2018) IS 9899:2018: Programming Languages — C. International Organization for Standardization, Geneva, Switzerland.
[11] Souyris J., Wiels V., Delmas D., Delseny H. (2009) Formal Verification of Avionics Software Products. In: Cavalcanti A., Dams D.R. (eds) FM 2009: Formal Methods. FM 2009. Lecture Notes in Computer Science, vol 5850. Springer, Berlin, Heidelberg.
[12] Efremov D., Mandrykin M., Khoroshilov A. (2018) Deductive Verification of Unmodified Linux Kernel Library Functions. In: Margaria T., Steffen B. (eds) Leveraging Applications of Formal Methods, Verification and Validation. Verification. ISoLA 2018. Lecture Notes in Computer Science, vol 11245. Springer, Cham.
[13] Leinenbach D., Santen T. (2009) Verifying the Microsoft Hyper-V Hypervisor with VCC. In: Cavalcanti A., Dams D.R. (eds) FM 2009: Formal Methods. FM 2009. Lecture Notes in Computer Science, vol 5850. Springer, Berlin, Heidelberg.
[14] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood (2009) SeL4: formal verification of an OS kernel. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP ’09). Association for Computing Machinery, New York, NY, USA, 207–220.
[15] Xavier Leroy (2009) Formal verification of a realistic compiler. Commun. ACM 52, 7 (July 2009), 107–115.
[16] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche (2017) HACL *: A Verified Modern Cryptographic Library. ACM Conference on Computer and Communications Security (CCS), Dallas, United States.
[17] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, Virgile Prevosto (2020) ACSL: ANSI/ISO C Specification Language, Version 1.15.
[18] Moy, Y. (2009) Automatic Modular Static Safety Checking for C Programs. Ph.D. thesis, Université Paris-Sud.
[19] Mandrykin, M.U., Khoroshilov (2016) A.V. Region analysis for deductive verification of C programs. Programming and Computer Software 42, 257–278.
[20] Xavier Leroy, Andrew Appel, Sandrine Blazy, Gordon Stewart (2012) The CompCert Memory Model, Version 2. [Research Report] RR-7987, INRIA. pp.26.