Rheinland-Pfälzische
Technische Universität
Kaiserslautern
Landau

Designing a Secure and Space-Efficient Executable File Format for the Unified Extensible Firmware Interface

Master's Thesis

by

Marvin Häuser

17th July 2023

University of Kaiserslautern-Landau
Department of Computer Science
67663 Kaiserslautern
Germany

Examiner: Prof. Dr. rer. nat. Klaus Schneider
Julius Roob, M.Sc.

Verwendete Hilfsmittel

Zur Erstellung dieser Arbeit wurden Grammarly, LanguageTool und DeepL Write als Hilfsmittel zur allgemeinen Verbesserung der Sprache (insb. zur Korrektur von Rechtschreib- und Grammatikfehlern) verwendet.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema „Entwurf eines sicheren und platzsparenden ausführbaren Dateiformats für das Unified Extensible Firmware Interface“selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit — einschließlich Tabellen und Abbildungen —, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 17.07.2023

Marvin Häuser

Marvin Häuser

Abstract

Executable files and libraries are at the core of operating systems and firmware implementations based on the UEFI PI and UEFI specifications, both functionally and security-wise. However, the UEFI PI and UEFI reference implementation EDK II has been subject to various reliability, maintainability, and security issues related to its PE image file loader. While many of the issues can be attributed to human error during development, the PE format and its associated Authenticode digital signature scheme add a great deal of complexity in their own right. Attempts to make the EDK II PE image file loader more secure have been successful, but maintenance and validation remain difficult. At the same time, there have been various problems with the TE format specified by the UEFI PI specification, which is a stripped-down variant of the PE format aimed at saving space on the firmware storage. Not only is it subject to its own set of design problems, but the format also does not leverage the full potential for space-saving.

We propose a novel executable file format, accompanied by a trivial digital signature scheme. Both help reduce the complexity of parsing and validation, while also encoding metadata much more efficiently than the PE and the TE formats. They are specifically designed for UEFI firmware implementations and are explicitly not optimized for the needs of modern operating systems. Compared to existing alternatives such as ELF and the Mach-O format, the proposed alternative makes extensive use of encoding techniques to impose certain constraints on the metadata, reducing the need for conformance validation.

Acknowledgements

Prof. Dr. rer. nat. Klaus Schneider, Chairman, and **Julius Roob**, Doctoral Student, both from the Embedded Systems Group at the University of Kaiserslautern-Landau, for supervising this thesis and continuously providing valuable feedback throughout the writing process.

Gerd Hoffmann, Software Engineer at Red Hat, for help in upstreaming prerequisite patches to EDK II to improve parsing security.

Ard Biesheuvel, Engineer at Google, and **Leif Lindholm**, Principal Engineer at Qualcomm, for providing important insights into the historical and current design decisions of the related EDK II infrastructure.

Pedro Falcato, Undergraduate Student at the University of Lisbon, for crucial research assistance with sparsely documented details of ELF files and the associated compiler and image file linker infrastructure.

Laszlo Ersek, Software Engineer at Red Hat, for invaluable guidance on EDK II and feedback on a new implementation of the EDK II image file loader.

Mikhail Krichanov, Doctoral Student at ISP RAS, for various contributions to the new executable file generation tool, in particular parsing of ELF files and refining the generation of PE files.

Savva Mitrofanov, Information Security Researcher at ISP RAS, for fuzz testing the PE implementation of the new EDK II image file loader library and providing the CI/CD infrastructure to run automated build and boot tests.

Vitaly Cheptsov, Lead Software Architect at ISP RAS, for countless years of guidance on software and firmware design, software testing and validation, and binary file formats and parsing in particular.

To Opa und Oma Rösch, Nani, and Annerose.

17th July 2023, Marvin Häuser.

Table of Contents

1. Introduction

Image files are core components of modern computer systems. They encode libraries and executable applications for efficient yet flexible consumption and execution, respectively, by an execution environment such as an operating system or a firmware implementation [1, 2, 3]. In order to support complex dependency trees, a responsive user experience, and advanced security features, the common image file formats have various powerful features [4, 5, 6], which come at the cost of complexity and increase the maintenance burden. The entire software stack involved — the image file loaders, dynamic linkers, and other processing tools — grows along with the image file format specifications they target. However, many of the use cases in the areas of embedded systems and firmware development do not require sophisticated features designed for the needs of advanced operating systems [2, 3].

To reduce the maintenance and platform resource burden of enabling such features, we propose a novel executable file format to be used by firmware implementations that conform to the UEFI PI and UEFI specifications [2, 3]. UEFI is currently the industry standard for firmware of personal computing devices running the Microsoft operating systems, Apple macOS on Intel architecture Macs, and many machines that ship with Linux or Android [7, 8, 9, 10]. In order to identify the necessary features to be supported by the new design, we will briefly introduce key concepts of the most common image file formats. This will help us to build a useful abstraction of UEFI executable files — a specific class of image files.

However, before we get into the technical details of image files, let us first lay out the foundations on which they are built.

1.1. Address Spaces and Virtual Memory

Address	0	1	2	3	4	5	6	7	8	9	10	11	12
Memory	‘H’	‘e’	‘l’	‘l’	‘o’	‘,’	‘ ’	‘w’	‘o’	‘r’	‘l’	‘d’	‘!’

Figure 1.1.: *Processor Main Memory Addressing.*

Example of main memory addressing for the string literal ‘Hello, world!’. Normally, main memory addressing is byte-wise, zero-based, and contiguous. In this case, we assume an encoding of one byte per character, so that each character resides in a dedicated memory cell.

Computer processors need memory to read from and write to instruction operands and results [11, 12]. In addition to the small processor registers, of which there are usually only a few, there is usually a much larger main memory. The processor indexes the main memory with addresses, which are non-negative integers and usually correspond to individual byte locations (see Figure 1.1) [1, 11, 12]. This is called physical memory addressing.

Note that some platforms allow memory-mapped input/output (MMIO) regions to be mapped into the physical address space [11]. In particular, the firmware storage device that holds the reset vector (the code that the CPU executes after a reset) is usually mapped into physical memory [11].

1.1.1. Absolute and Relative Addressing

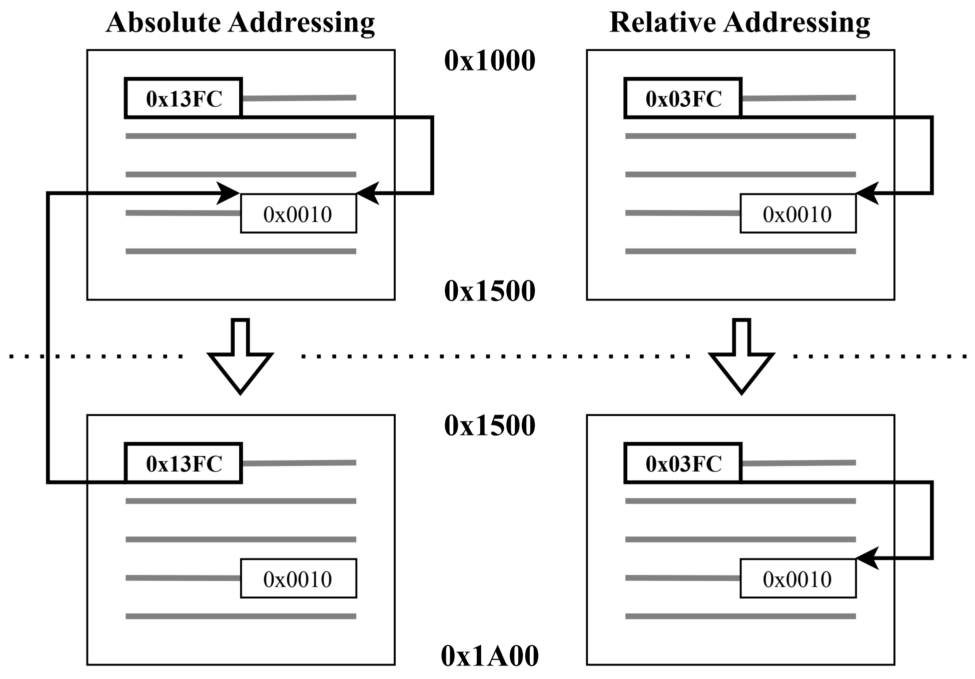

Figure 1.2.: *Absolute and Relative Addressing.*

Two memory blocks with internal self-references. The left one uses absolute addressing, i.e. instructions store the address of the referenced datum, while the right one uses relative addressing, i.e. instructions store the offset from their addresses to the address of the referenced datum. The bottom half of the figure shows what happens when moving both blocks from the address 1000_{16} *to* 1500_{16} *without any further modification. The absolute address reference still points to the old location of the datum, while the relative address reference points to its new location.*

For sophisticated computer processors (e.g. those using the Intel or Arm instruction set architectures (ISAs)), there are typically two classes of addressing modes [11, 12]. Absolute addressing uses memory addresses as described above. Relative addressing, on the other hand, uses memory address offsets that are added to the current instruction address. The advantage of the latter approach is that, if a self-contained memory block referencing its internal data (e.g. an image) is moved to a different start address, this invalidates all internal self-referencing absolute addresses, while all relative addresses remain valid (see Figure 1.2). To move memory referenced by an absolute address, it must be updated before dereferencing it again (see Section 1.5.5).

1.1.2. Memory Pages and Virtual Address Spaces

To allow for advanced memory management, an abstraction called virtual memory is built on top of physical memory [1]. The virtual address space is typically segmented into memory pages, each of which can be individually backed by a different physical memory page, or not be mapped at all (see Figure 1.3). Accesses to unmapped virtual memory pages usually result in a processor exception that can be caught by the operating system. This allows virtual memory to be populated on demand with data from secondary memory, such as persistent storage. Memory swapping is a technique that uses this concept to temporarily offload data from the main memory to persistent storage, particularly in high memory contention situations [1, 13].

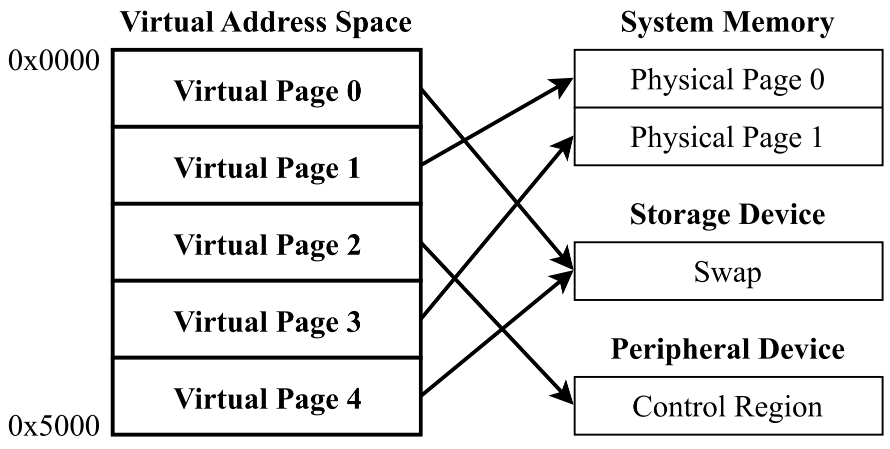

Figure 1.3.: *Virtual Addressing.*

Example of a virtual memory mapping. As shown, virtual memory pages can be mapped to any physical memory page in any order, to a peripheral device MMIO region, or to secondary memory such as the memory swapping device. In the latter case, the virtual memory page may be technically unmapped, and the operating system backs it on demand when handling the page fault exception.

1.2. Memory Allocation Schemes and Memory Management

Firmware implementations and operating system kernels manage platform resources to, among other things, provide drivers and applications with sufficient memory. There are three main approaches to memory allocation. Static memory allocation reserves fixed amounts of memory with a static lifetime, i.e. it is allocated for the duration of program execution [14, 15]. Call stack memory allocation allocates (usually small and fixed amounts of) memory with a local lifetime, i.e. it is allocated exactly while in scope, and it is automatically freed when the scope ends. For this reason, C also refers to call stack frame variables as 'automatic variables' [14]. In contrast, dynamic memory allocation reserves (usually large or flexible) amounts of memory with a dynamic lifetime, i.e. it is not limited to a simple scope-based lifetime. C requires manual memory management for dynamically allocated memory [14], while the safe subset of Rust does not support it at all [15].

1.2.1. Static Memory Allocation

For global data, this is done as part of the initialization of the program and then remains valid throughout its lifetime [14, 15]. Essentially, the compiler reserves the memory within the image memory (see Section 1.5.2), there is no runtime memory allocator involved and so there is no room for memory allocation errors.

The size of statically allocated memory is constant, so compilers and code analysis tools know its bounds [14, 15]. Because of the static lifetime guarantee, it is not possible to create references to statically allocated memory that may become invalid at runtime. However, references can still become invalid if statically allocated data is available externally, e.g. an exported global variable of a shared library, and the consumers retain references to it after the producer has terminated.

1.2.2. Call Stack Memory Allocation

The call stack requires incremental memory allocation for call stack frames, i.e. the areas required by specific subroutine calls (see Figure 1.4) [1, 11, 12]. Unlike static memory allocation, call stack memory allocation requires a memory allocator. The program itself embeds its logic, expanding or shrinking its call stack in a last-in-first-out fashion as needed. While the individual call stack frames typically have fixed sizes (variable-length arrays are a rare exception), unbounded recursions can require infinitely many call stack frames and thus infinitely much call stack memory (except in cases of e.g. tail-recursion elimination [16]).

Memory allocation errors are possible when running out of call stack memory, although software engineering best practices aim to make this unlikely [18]. With known upper bounds on all recursions and in the absence of variable-length arrays with unknown upper bounds, a reasonable upper bound on the call stack memory requirement can be computed. This allows the call stack size to be chosen accordingly to ensure that it is sufficient for worst-case executions. Unlike with static memory allocation, references can become invalid at runtime, if they are live when their scope ends [14]. A common example of this is returning a reference to a local variable.

```
void f(void) {
  uint8_t  f1;
  uint32_t f2;
}

void g(void) {
  uint64_t g1;
  f();
}

void h(void) {
  g();
}
```

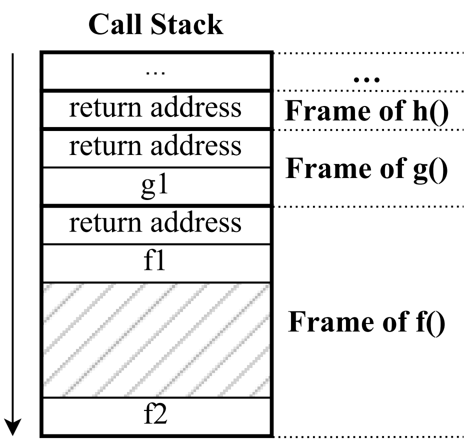

Figure 1.4.: *Call Stack and Call Stack Frames.*

A source code example and the corresponding call stack layout. The call stack generally grows from top to bottom in a last-in-first-out fashion and consists of successive call stack frames of subroutine calls. The data within a call stack frame are aligned according to the application binary interface (ABI) alignment requirements, i.e. the compiler is likely to insert data structure padding between `f1` and `f2` (the illustration assumes natural data alignment). The call stack frames may have stricter data alignment requirements [17], e.g. to satisfy the requirements of vector instructions [11] on call stack memory. In a conventional control flow, each call stack frame contains the return address of the instruction following the subroutine call.

1.2.3. Heap Memory Allocation

Heap memory allocation, also known as dynamic memory allocation, reserves memory with a dynamic lifetime from heap memory [1, 14, 15]. Dynamically allocated memory remains valid regardless of scope (including the entry point of the program for some environments) and must be manually freed. It is typically used with variable-size or large amounts of memory. Unlike the schemes discussed previously, dynamically allocated memory is not necessarily live until the end of its scope (i.e. the end of the block in which it was declared), so intermediate memory can be freed as needed [14]. Also, subroutines can return dynamically allocated memory, transferring ownership of it without creating a copy. These points can contribute to lower peak and average memory consumption. On the other hand, dynamic memory allocation requires a complicated memory allocator (e.g., the SLUB allocator [19]). By its very nature, heap memory tends to become fragmented over time, which can reduce performance and efficiency for future allocations [19]. To mitigate these problems, real-world operating system algorithms attempt to reduce and avoid fragmentation through means such as overallocation.

There is no general method to derive reasonable upper bounds for memory consumption among other metrics with dynamic memory allocation. Even if there were, heap memory fragmentation can gradually reduce the effective space available. Thus, it is generally not feasible to provide worst-case guarantees, which is why some embedded systems prohibit dynamic memory allocation altogether, especially when they have to operate in real-time [20]. References to dynamically allocated memory may become invalid if they are live after the memory has been freed [14].

1.2.4. Memory Safety Violations

Because memory is a shared medium, any memory safety violation can have serious consequences. Especially in kernel-space or firmware contexts, rogue reads can expose secrets and rogue writes can hijack the control flow. The following is a brief description of the main classes of memory safety violations.

General Memory Safety Violations

All memory allocations have fixed bounds and are subject to the same data type rules. As such, there are common memory safety violations that are independent of the memory allocation scheme.

- **Buffer overflow:** Accessing a memory buffer outside its allocated range. This may result in reads or corruption of subsequent memory.

- **`NULL`-pointer dereference:** Dereferencing a `NULL`-pointer. While this can happen in any context, dynamic memory allocation is one of the most important sources of valid `NULL`-pointers [14]. In particular, low-level software may map the memory page containing the `NULL` address [21]. In this case, the consequences are similar to use-after-free (see Section 1.2.4). The inventor of the `NULL`-pointer, Tony Hoare, has called this his 'billion-dollar mistake' [22].

Call Stack Memory Safety Violations

Call stack memory allocation, despite having no dynamic layout logic for its allocation scheme, features a unique safety violation. This is particularly relevant for C programs, as the compiler automatically manages the allocation and lifetime of call stack variables, but does not provide any safety guarantees.

- **Use-after-scope:** Accessing a call stack variable after its scope has ended. Due to automatic scope-based memory management, this immediately invalidates any reference to it.

- **Call stack overflow:** A buffer overflow with the call stack memory. This is special because the call stack stores the return address for the current function call. Successfully overwriting it may allow the execution of arbitrary code, depending on the platform's security mitigations.

Dynamic Memory Safety Violations

Dynamic memory allocation safety violations are among the most common and critical vulnerabilities in the software industry [23]. For C programs, it is particularly prone to human error due to manual lifetime management and limited analysis capabilities of the compiler.

- **Use-after-free:** Accessing a reference to previously dynamically allocated memory. If the address has been reused for new dynamically allocated memory, accessing the reference may read or corrupt unexpected memory.
- **Double free:** Freeing a reference to previously dynamically allocated memory again. Depending on the implementation of the dynamic memory allocator, this may corrupt its metadata.

1.3. Malware and Software Security

The classic example of a software attack is malware, which must be actively run on the target computer. It can freely execute code and can perform any action that is allowed within the user-space, or potentially the sandbox, in which it was executed. While this is one of the simplest methods of a software attack, it requires conscious user interaction and is limited by the privileges of the environment in which it is invoked. In the past, malware had a relatively simple way of getting into the kernel-space by asking the user to grant elevated privileges and install a kernel-space driver. However, modern operating systems such as Windows, macOS, and Linux derivatives support kernel-space driver digital signature authentication (see Section 1.5.7) in combination with Secure Boot (see Section 3.2.5). This means that user-initiated privilege escalation alone is no longer sufficient to penetrate the kernel-space. Recent versions of macOS on Apple silicon even require the user to enable loading of third-party kernel-space drivers by booting into a special recovery environment, regardless of the driver's digital signature [24].

Despite the best efforts to improve the security of operating systems and firmware, UEFI malware still appears from time to time [25, 26]. To help combat this growing threat, we must first understand the very basics of common software exploits.

1.4. Memory Exploits

Software exploits are a way of further attacking the kernel-space and are usually stealthy. Rather than tricking the user into granting elevated privileges, they aim to trigger bugs in software (including kernel-space drivers) that have or are expected to gain elevated privileges. A common family of attack vectors are the memory safety violations (see Section 1.2.4) discussed earlier. A particularly interesting development in attacks is the move towards return-oriented programming (ROP).

One of the first sophisticated and widespread attacks was call stack smashing, a technique for corrupting the call stack with attacker-controlled data [27]. Using call stack overflows, it was previously possible to place shellcode on the call stack. By overwriting the return address of the current call stack frame, the attacker could direct the control flow to that shellcode to execute malicious code. In response, operating system kernels marked all call stack memory as non-executable (see Section 3.1.2).

Return-to-libc is a spiritual successor proposed to work around the non-executable call stack. It still relies on overwriting the current return address, but instead of jumping to code controlled by the attacker, it targets code already in memory, so-called code-gadgets. One of the most common targets was libc, as it is loaded into the address space of almost all user-space programs. As part of the demonstration of such an exploit, the reporter proposed the mitigation to fix call stack overflow attacks based on zero-terminated American Standard Code for Information Interchange (ASCII) operations [28]. However, it was successfully bypassed by using the PLT, which was not affected by the mitigation [29].

As the defences grew stronger, the attacks became more versatile, expanding their scope beyond libc to other common shared libraries as a source of code-gadgets [27]. In order to exploit even small memory safety violations, sophisticated techniques have been developed to chain code-gadget calls. These are now known as ROP.

While other families of memory exploits exist, such as heap memory and format strings attacks, they are not fundamentally different from the techniques discussed here. However, they dramatically expand the attack surface for memory exploits. Research conducted by leading low-level security teams has shown that memory exploits are by far the largest class of serious security vulnerabilities [27, 30, 31, 32].

1.5. Image File Formats

Images capture memory content, layout, and state. They make it easy to bootstrap the program part of a process address space [33]. The executable part of an image carries the program code, while the data part supports static memory allocation, as discussed earlier (see Section 1.2.1). Because the common image file formats serve very similar purposes, there have been various design conventions and common concepts between them. These cover both the data structures used to describe the components of and the operations on the image file and image address space itself. From them, a kind of abstraction can be sketched as follows.

1.5.1. Image File Parsing and the Image File Header

Image files are generally header formats, meaning that they start with a data structure called image file header, which contains information on how to interpret the rest of the file (see Figure 1.5) [33]. This includes information such as the image base address (see Section 1.5.5), the offset of the image file section table (see Section 1.5.3), and the offset of the image segment table (see Section 1.5.2). To easily distinguish between file formats, the image file header usually starts with a multi-byte file magic number, a constant value to uniquely identify the format. There are also usually means to support newly introduced features while maintaining backward compatibility, such as version numbers, feature flags, and data structure size information.

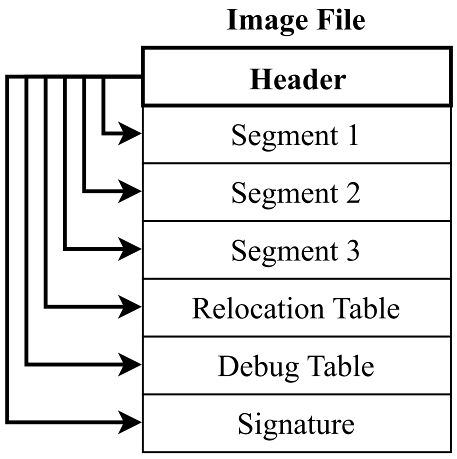

Figure 1.5.: *Image File Header.*

The image file header is the data structure at the beginning of an image file that provides information about the rest of the data. It usually identifies the image file format with a file magic number. From there, it indexes image segments, image file sections, the image relocation and debug tables, and optionally the digital signature.

In general, image file format designs work in such a way that they can be parsed by interpreting their control data as C structs [4, 5, 6]. Using pointer or address arithmetic, the various offsets in the image file header can be used to access different data structures containing information on how to interpret or process the data. In the following, we will discuss image file loading, static linking, dynamic linking, and image relocation as examples of operations on image files and images, along with the data structures that enable them.

1.5.2. Image File Loading and Image Segments

Images owe their name to the fact that they capture the content and state of memory [33]. The main tool for this purpose is image segments (for PE files, image file sections and image segments are a joint concept [6]). They describe how to set up both the content of the image address space (e.g. copy or map memory from the image file or initialize with zero-values) (see Figure 1.6) and its memory permissions [4, 5, 33]. As such, they are most relevant to image file loading and execution.

Historically, the two most important image segments are `text` and `data`. The former conventionally carries executable code, while the latter holds

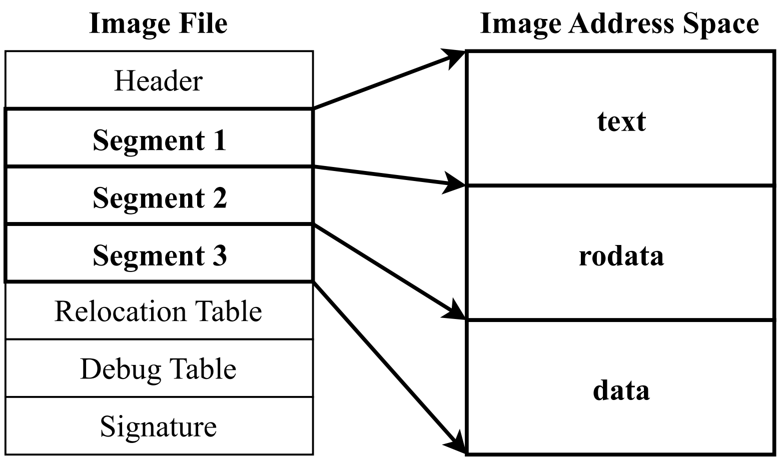

Figure 1.6.: *Image Segments.*

Image segments compose the entire image address space, both in terms of content and state (e.g. memory permissions). In general, they do not logically organize content, but group it by their kind (e.g. executable code, immutable data, mutable data, etc.).

non-executable data, in particular statically allocated global variables (see Section 1.2.1) [4, 5, 33]. In addition, `bss` stores uninitialized global variables (implicitly defined as a zero-value in C [14], among other languages) and generally does not occupy space in an image file, because it is initialized to all zero-values at load-time. Finally, to improve memory security, `rodata` was introduced to mitigate exploits such as overwriting format string literals by enforcing immutability. An alternative approach is to include read-only data in the `text` image segment (see Section 2.1.4), which is also read-only (see Section 3.1.3).

Image file loading is the process of transforming an image file into an image address space [33]. In conjunction with virtual memory management, modern operating systems usually just memory-map the image segments from the image file on the secondary storage into the process address space. Image segment components that are modifiable, such as GOT, work via copy-on-write (COW) semantics. However, firmware implementations, most notably UEFI, generally do not support virtual memory management and instead copy the data explicitly into the main memory [2, 3, 21].

To efficiently compose image segments, image file linker use image file sections to categorize different types of data and code [33]. We will explain this composition process in more detail below.

1.5.3. Image File Static Linking and Image Sections

Image file sections are units to logically organize the data of an image (see Figure 1.7) [4, 5, 6, 33]. For ELF, Mach-O, and COFF files, this can be a very granular organization, depending on the build toolchain. For example, Mach-O executable files generated by the Apple Xcode toolchain have a dedicated `cstring` image file section for string literals [5]. ELF compilers may generate object files that go as far as having an image file section per function and global datum to enable dead code and data garbage collection and improving locality [34]. Image files of both formats generally have image file sections for image relocation fixup and image symbol table information [4, 5]. For PE files, the organization is generally not granular, as they double as image segments (see Section 1.5.2) [6].

Static linking combines several image files into a joint library or executable file [33]. As suggested earlier, the image file linker can use information about references to an image file section to garbage-collect dead code and data. Not considering the specific configuration, static linking essentially merges all data structures, including image segments, image file sections, and image relocation fixups. One use case is to include static libraries, which is a consumable collection of useful functions and data, within an executable file.

One notable implication is that, if multiple executable files consume the same function or datum, it will be duplicated across the processes' address spaces [33]. To mitigate this, where beneficial, we will now introduce the concept of dynamic linking.

1.5.4. Image Dynamic Linking and Image Symbols

User-space programs usually share a lot of code. The most prominent example is libc [34], but operating systems also provide various application programming interfaces (APIs) to handle user, hardware, and software interaction. To include this shared code in each program individually would result in huge binaries and potentially cause caching problems, as frequent accesses to the same code and data technically target different memory.

To efficiently share code between programs, shared libraries complement static libraries and dynamic linking complements static linking [33]. Essentially, only one copy of the shared library is loaded into the main memory, and any dependent images do not resolve their dependencies on it at build-time, but at load-time or runtime. This works by indexing image symbols, which are essentially named addresses of functions and global data. Dependencies can instruct the dynamic linker to resolve the address of the symbol and adjust the image memory to reference it, similar to image relocation fixups. Typically, a common data structure can describe both dynamic linker instructions within the image relocation table [4, 5].

Dynamic linking can be done eagerly or lazily [33]. In the former case, the load time of the program increases, because the dynamic linker resolves all image symbols even if they may never be accessed during the lifetime of the image. In the latter case, function references can be resolved the first time they are used by implementing a stub indexed by the PLT. However, this requires the resolution targets to be writable at runtime. Thus, high-security configurations typically force the use of eager dynamic linking.

Since every executable address space is different, there is no way to guarantee that the image base address of a shared library will be available across all of its consumers [35]. Furthermore, avoiding predictable addresses can be a powerful

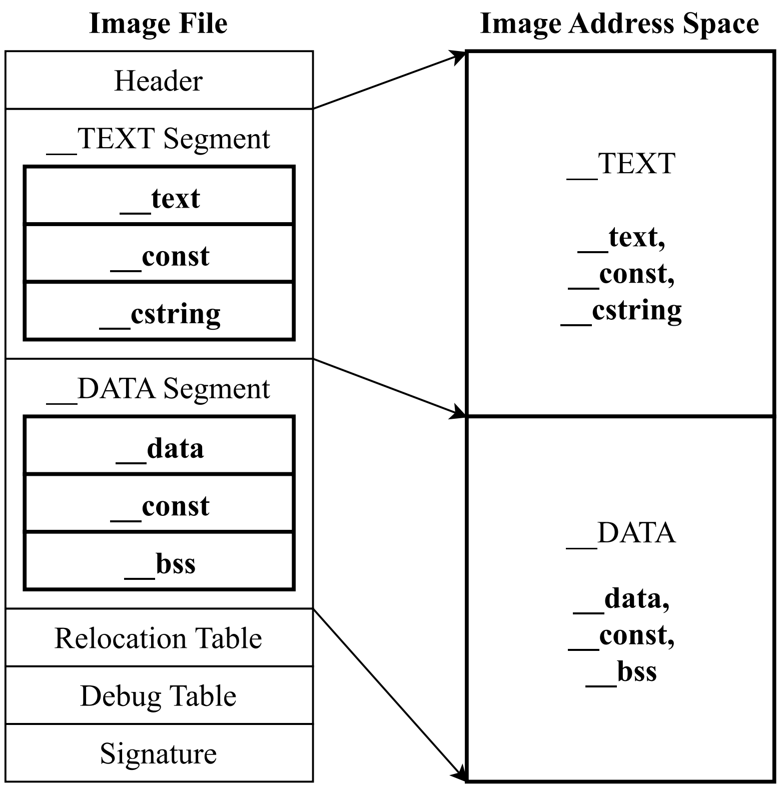

Figure 1.7.: *Image File Sections.*

The data that make up an image segment are bundled into chunks managed by image file sections so that the image file linker can flexibly move, rearrange, and merge them at link-time. This has several advantages, from garbage collection of unreachable code and unreferenced data to optimization of the binary layout.

tool for making memory exploits less reliable. The solution to both is image relocation, which we will discuss next.

1.5.5. Image Relocation and Position-Independence

Modern operating systems have complex requirements for the dynamic loading of shared libraries, drivers, and applications. This includes the ability to load images at arbitrary addresses (see Section 1.5.4). This can also be used to increase resilience against various software exploits by using pseudo-randomization for the load address of images as part of Address Space Layout Randomization (ASLR) [36]. Binaries that support being moved are commonly referred to as position-independent code (PIC) [33]. Not all binaries are inherently position-independent, as the compiler may generate CPU instructions or data with absolute addresses to data within the image. This may happen because the target ISA has poor support for relative addressing, absolute addressing may be more performant, or there are global pointers-to-pointers initialized at build-time.

A common part of solving the problem of position dependency is the indexing and processing of image relocation fixups (see Figure 1.8) [33]. In this context, they are instructions to the image file loader on how to modify CPU instructions and addresses to execute from a specific address. The dynamic linker links the image to run at an address specified at build-time, commonly referred to as the image base address. CPU instructions and addresses that require absolute addressing to data in the image get image relocation fixups indexed, which tell the image file loader where and how to update them. CPU instructions that do not trivially encode absolute addresses get fixed by image relocation fixups that are specific to the host architecture declared by the image file header [4, 6, 37]. The image file loader then processes the image relocation fixups as part of image file loading by adding the offset of the image base address and the final load address to the relocatable address in the manner required by the image relocation fixup type [33]. In a different context, static linking also uses image relocation fixups, but they are typically indexed differently (i.e. per image file section rather than for the whole image) and have different semantics (e.g. for relative addressing across image file sections).

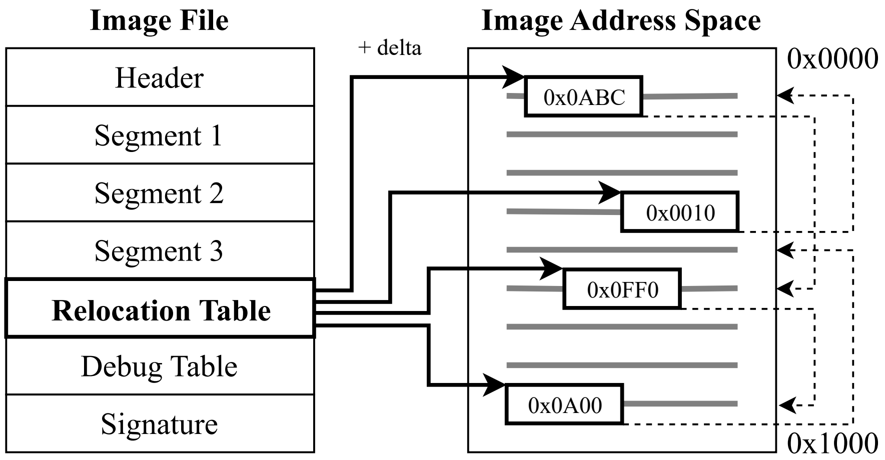

Figure 1.8.: *Image Relocation Fixups.*

The image relocation table contains image relocation fixups which tell the image file loader how to update the image address space when moving it from the image base address (where 'delta' is their difference). The dashed lines visualize the absolute addresses that the image relocation fixups target within the image address space.

UEFI has adopted this model of dynamic image file loading to meet its own needs for dynamic driver dispatching and execution of arbitrary applications at operating system runtime [2, 3]. It has also extended it with the notion of runtime image relocation. This allows the operating system to map memory required by UEFI during the operating system runtime to non-identity virtual memory addresses. It enables Kernel Address Space Layout Randomization (KASLR) to be extended to include UEFI runtime memory [8].

1.5.6. Additional Metadata and Extensibility

Beyond the essential data, image file formats often support some form of metadata extensibility [4, 5, 6]. This can be debugging information, for example, to help developers with the development process. In general, most image files formats support special image file sections with predefined names. However, this requires the metadata to be mapped into the image address space, which is not always desirable. Therefore, some formats provide maps of identifiable data structures, which may or may not point to data in the image address space [5, 6]. Some also define image file header data structures with flexible sizes to ensure extensibility and backward compatibility [4].

1.5.7. Image Authentication and Cryptographic Hashes

For security reasons, operating systems and firmware implementations usually authenticate image files. This ensures that the image file has been approved by a trusted entity and has not been tampered with by a third party. This is achieved by using digital signatures, which leverage asymmetric cryptography to securely sign the image files. The public key in this scheme can be used to verify that the digital signature is correct and that the image files is authentic. Image authentication is the basis of UEFI Secure Boot and Measured Boot (see Section 3.2.5).

There are two approaches to image authentication. The first is to digitally sign the image file itself. This is a simple scheme that ideally cryptographically hashes the entire file in one piece (see Figure 1.9).

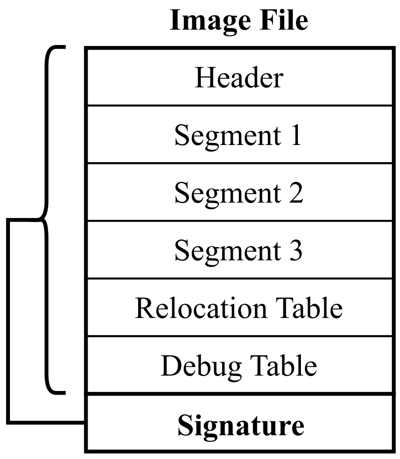

Figure 1.9.: *Digital Signature based on the Image File.*

An image authentication scheme that digitally signs the image file itself. It may cover the entire image file or only a subset of it (e.g. see Section 2.2.3). However, it must cover the image address space implicitly, i.e. all image segments, to be secure.

Hashing the image address space (see Figure 1.10) has the advantage that it integrates well with the virtual memory approach to image file loading and memory swapping. Technically, individual image memory pages can be lazily loaded many times throughout the lifetime of the image. To prevent time-of-check to time-of-use (TOC/TOU) attacks, they should ideally be authenticated each time they are fetched from secondary memory. One approach to this is to store a data structure containing cryptographic hashs for each memory-mapped memory page of the image file and then digitally sign it. Now, each time a memory page is fetched, its cryptographic hash can be compared to the in-memory copy of the data structure copy to authenticate it quickly and lazily. Since the entire image file is usually memory-mapped, which is pretty much mandatory to take full advantage of virtual memory in this context, this implicitly authenticates the entire image file.

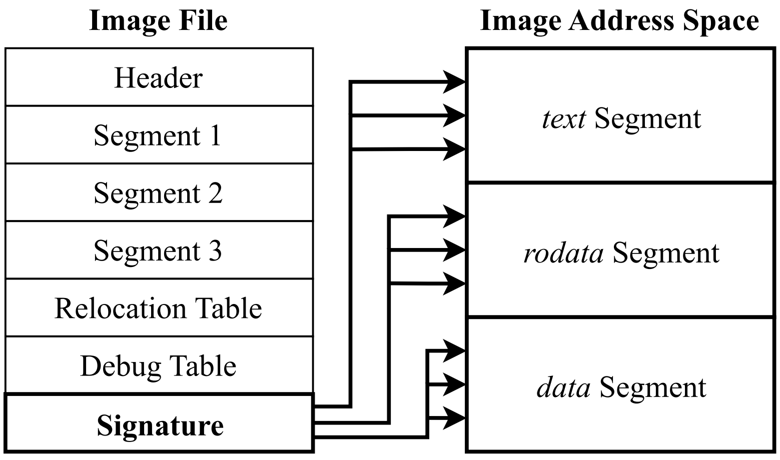

Figure 1.10.: *Digital Signature based on the Image Address Space.*

An image authentication scheme that digitally signs the image address space directly. To enable lazy loading and thus authentication of image memory pages, it is particularly useful to store a map of their cryptographic hashes. If the entire image file is memory-mapped, as is typical for image file loading powered by virtual memory, it is implicitly authenticated as a whole.

Similar concepts of nested cryptographic hashing and lazy authentication can be found in the field of filesystems [38]. Because they have a hierarchical structure rather than a flat one, it is desirable to be able to lazily authenticate each level on demand. This is done not just by using two levels of cryptographic hashs, but by computing a tree of them, where a parent hashes the aggregated hashes of all its children [39]. This is known as a Merkle tree, a generalization of the nested cryptographic hashing concept.

1.6. Problem Statement and Proposals

As we have seen, memory safety violations are a significant and critical threat to system security and parsing in particular is a likely source of them. Current UEFI firmware provides only a few of the state-of-the-art defences employed by operating systems. At the same time, there are tight constraints on the platform code size, such that space efficiency is a valid concern. The UEFI Forum's previous attempt at a new executable file format in the form of TE failed to address both security and space efficiency concerns with the existing PE specification and infrastructure (see Section 2.3.1) [40].

In addition, image file modification and conversion are crucial for building UEFI PI and UEFI software, especially for the EFI Development Kit II (EDK II) build system (see Section 2.5). However, the existing tools all rely on direct conversion between formats and do not have sufficient testing in place. As a result, the generated image files are often malformed or insecure [41, 42].

In response, we propose a novel executable file format to reduce the attack surface for parsing errors and bugs, as well as the complexity of the parser itself, while also rethinking the encoding of the metadata to increase the space efficiency. To generate files of this format, we also propose a novel executable file conversion stack that relies on an intermediate representation to abstract and validate the conversion process.

1.7. Outline

After this brief introduction to images, their basic concepts, and the importance of secure parsing, we will now give an outline of this thesis.

- First, we will discuss related work in Chapter 2. In particular, we are interested in common real-world image file formats and their design in the context of modern operating systems, our target environment UEFI, and related safety and security practices.
- Chapter 3 begins our contribution with requirements engineering for our proposal of a novel UEFI executable file format. We will then define an abstraction of UEFI executable files from its results and common practices across image file format designs.
- We will detail design considerations that combine industry best practices in encoding and parsing with the requirements and guarantees of UEFI, for such a novel format in Chapter 4.
- In Chapter 5, we will explain the architecture and principles of the implementation, together with our testing techniques and methodology.
- Finally, we will conclude the thesis with our results in Chapter 6, our conclusion in Chapter 7, and future work in Chapter 8.

2. Related Work

To get a good understanding of the basics of images and image file formats, we will first discuss common formats used across different operating systems and firmware implementations in Section 2.1. We will then look at different types of image authentication schemes in Section 2.2. Next, we will review the state of the art for the related UEFI technologies and the reference implementation in Section 2.3. After, we will briefly outline the problems with current image file solutions and showcase an earlier attempt to address them in Section 2.4. In Section 2.5, we will go through existing projects for image file format conversion and what they have in common. Finally, in **??** and section 2.7, we will look at techniques for stress-testing for and guaranteeing memory safety.

2.1. Common Image File Formats

For workstations, both home and commercial, and servers, the most common operating systems are Microsoft Windows, Apple macOS, and operating system distributions based on the Linux kernel. In the following, we will introduce their respective preferred image file formats and their basic design ideas.

2.1.1. Executable and Linkable Format (ELF)

A common image file format for UNIX- and Linux-based operating systems is the Executable and Linkable Format (ELF). Formerly specified by the Tool Interface Standard (TIS) Committee [4], it is now managed by The Santa Cruz Operation, Inc. (SCO) as part of the System V and other ABI specifications [37, 43, 44]. Architecture-specific details of the format, such as specific image relocation fixup types, are defined separately in the ISA-specific ABI specifications [37, 44, 45].

The design of the image file format suits the needs of static linking, dynamic linking, and execution. The different views for the two image linking scenarios can be seen in Figure 2.1.

Position-Independence, GOT, and PLT

Several executable files can use the same shared library. It must therefore be mapped into all of their respective process address spaces. Immutable image segments will remain the same in all of them. For this reason, operating systems usually load these image segments only once and map them into all consumer process address spaces [35, 8, 10]. The Linux kernel in particular makes extensive use of COW, a technique for sharing code and data as immutable across arbitrarily many consumers and duplicating it when written to, for such

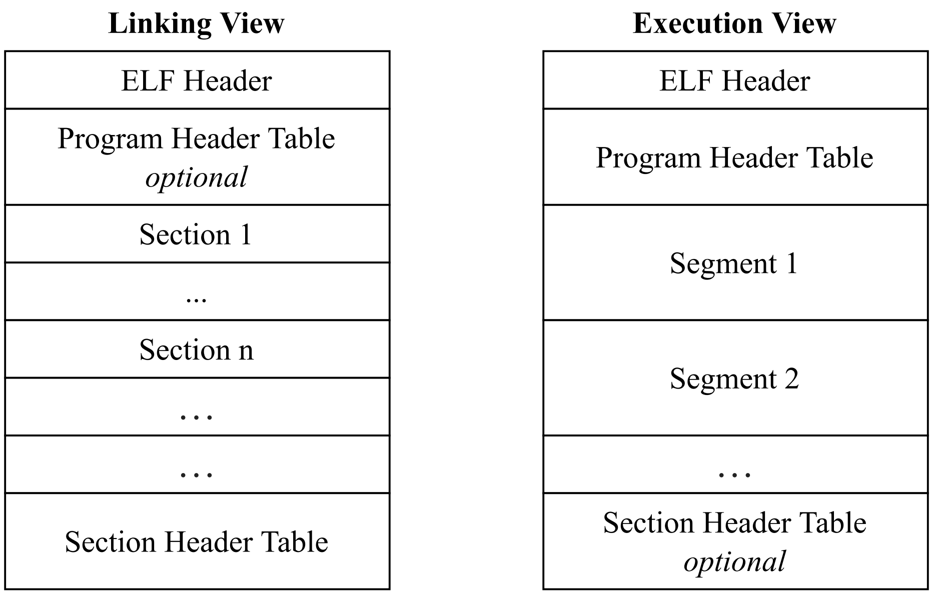

Figure 2.1.: *'Object File Format' (Figure 1-1.) [4].*

The ELF file views for image linking compared to execution. Image linking is primarily concerned with image file sections, the smaller building block for image files. Image file sections generally categorize types of content. Often, categories such as 'debug data' and 'string literals' are grouped into a common image file section. For object files, they may only cover a single function or datum. Image loading and execution work with image segments, the larger building blocks that are made up of image file sections. Image segments generally categorize data into memory permissions groups. For example, this may be 'executable code' or 'read-only data'. The content of the data is not relevant to image execution.

purposes [10]. For process address spaces, this is implemented by mapping the memory page as read-only and hooking the page fault handler to duplicate it.

Technically, read-only image segments can still be targeted by image relocation fixups. This will cause the same problems as described above for writable image segments and may duplicate many to most of the associated memory pages via COW. To reduce this amount of memory pages, the global offset table (GOT) was introduced as an indirection for memory accesses to externally linked data (see Figure 2.2) [33, 4]. By using relative addressing for GOT accesses, this makes these image segments truly immutable, and thus they will never be subject to COW duplication.

The procedure linkage table (PLT) follows the same ideas as the GOT, but it specifically targets executable code [33, 4]. As an indirection, the PLT carries stubs for all affected subprocedures.

RELR and RELRO

In recent years, RELR has established itself as the most efficient encoding for relative image relocation fixups in ELF to date [46]. The idea is to store

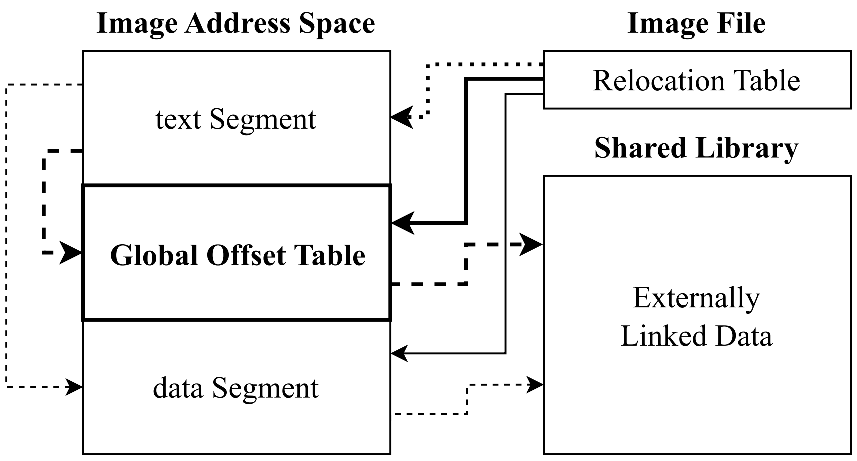

Figure 2.2.: *Image Global Offset Table.*

The GOT is an indirection layer to avoid process-specific image relocation fixups to read-only image segments. In particular, it is an indirection for all image memory accesses via absolute addresses. In the case of position-independent executables (PIEs), all accesses to GOT itself are relative. Different processes can map the same shared library to different load addresses. To account for this, each process has a copy of the GOT with the absolute addresses corresponding to its own address space. The dotted line visualizes the image relocation fixups that allow read-only image segments to address GOT or other data strictly within the own image address space (applies only to non-PIEs).

the address of a relocation target along with a bitmap of adjacent locations to relocate. The idea is to take advantage of the locality of relative image relocation fixups, e.g. such as those issued for global pointer arrays.

As mentioned before, lazy dynamic linking requires the image relocation targets to be writable, which may allow attackers to hijack them (see Section 1.5.4). ELF's solution to this is relocation read-only (RELRO), which, when fully enabled, forces eager dynamic linking and marks GOT and PLT as read-only when done [47]. Partial RELRO is supported, but not recommended for security reasons.

2.1.2. Mach Object File Format (Mach-O)

The default image file format for Apple operating systems, such as macOS, iOS, and watchOS, is the Mach Object File Format (Mach-O) [5]. At a high level, it is very similar to ELF, especially regarding PIC, which is the default mode for the Apple Xcode toolchain. However, unlike ELF, which requires features such as RELRO, initial and maximum protection levels are part of the Mach-O core design for image segments.

The traditional format for storing image relocation fixups in an image file is a table. It can be nested, so that there is a sub-table for each memory page of the image, and each such entry has a table of offsets relative to its memory page. Apple has introduced a new storage format for Mach-O in recent years, called dyld chained fixups [48]. This is again based on the table format with one sub-table for each memory page. However, the new format stores image relocation fixups as a linked list within an image segment at the image relocation target location (see Figure 2.3). This means that the target of an image relocation fixup stores a data structure that compresses its metadata, as well as the offset to the next image relocation fixup.

The Mach-O format introduced a constraint on the image address space size of 64 GiB, which requires $\log_2(64 \cdot 1024^3) = 38$ bits to store a valid address in an image address space if its image base address is 0. Meanwhile, since the chain is per memory page, the offset to the next image relocation fixup can be at most 4095 bytes, thus requiring $\lceil \log_2(4095) \rceil = 12$ bits to store. Both data require 50 bits of storage, which easily fits into the original 64-bit target.

This new organization of image relocation fixups can save a significant amount of storage space for image files with many image relocation fixups, as the ‘free storage space’ for chained image relocation fixups far outweighs the per-memory page tables of the image relocation table [48].

Furthermore, their design addresses an issue of temporally separate accesses to the same memory pages, which may require memory swapping during dynamic linking [48]. Namely, image relocation and dynamic linking are unified as a single step for each image relocation fixup, requiring only a single pass over each image segment memory page.

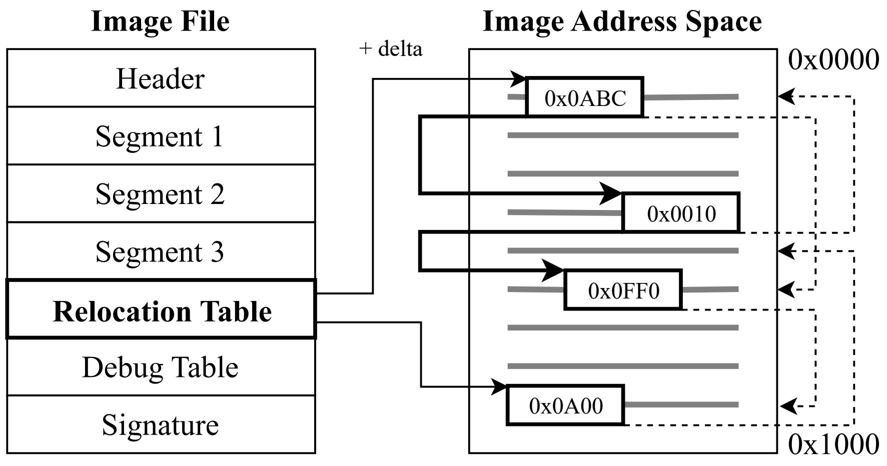

Figure 2.3.: *Image Relocation Fixup Chains.*

Similar to Figure 1.8, but singly-linked chains organize the image relocation fixups. As all the necessary information resides in the image relocation target, this helps to reduce the size of the image file and decreases the number of memory accesses outside the same memory page. This is implemented by the Mach-O format for image relocation fixups of the same type within the same memory page.

2.1.3. Portable Executable and Common Object File Format (PE/COFF)

For the Microsoft Windows operating system, Microsoft primarily uses the Portable Executable and Common Object File Format (PE/COFF) (see Figure 2.4) [6]. It is also the primary format specified by the UEFI PI and UEFI specifications [2, 3]. Unlike ELF and Mach-O, PE/COFF combines the concepts of image segments and image file section under the latter's name [6].

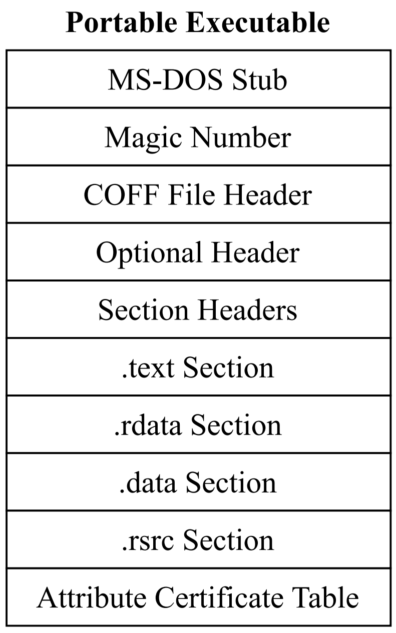

Figure 2.4.: *Portable Executable File Organization.*

Example PE file layout. The MS-DOS stub is virtually unused and is only a relic for backward compatibility. Due to the dual nature of PE and COFF files, there are separate data structures to describe metadata for both use cases, namely the Optional Header and the COFF File Header. The multiple layers of headers add to the complexity of PE parsing [40, 49]. The image file sections double as image segments. Optionally, there is a trailing certificate table that contains digital certificates.

To compress the image relocation directory, PE introduced the concept of Base Relocation Blocks [6]. They consist of a 4 KiB aligned base address within the image address space and a variable-size list of Base Relocations (image relocation fixups). By indexing the image relocation fixups per 4 KiB block, they only store their offset within the block, which requires only 12 bits instead of the 32 or 64 bits needed for a full address.

Caveats

PE suffers from dangerous data duplication, e.g. it has several `SizeOf*` fields, such as `SizeOfImage`, with sums of certain or all kinds of image segments sizes [6]. Relying only on their values alone may cause buffer overflows if they do not match reality [40]. However, to validate them, one needs to recompute the values from the image segment information. Therefore, for many use cases, it may be best to ignore these fields, as done by the Acidanthera UEFI Development Kit (AUDK) PE loader [49].

There are also unintuitive design choices that confused reviewers of the 'Securing the EDK II Image Loader' publication [40]. For example, the number of image relocation fixups within a block must always be even [40]. This is, because image relocation fixups information has a data alignment requirement of 2 bytes, while blocks must start on a 4 byte boundary [6]. An odd amount of image relocation fixups must be padded with another of type `IMAGE_REL_BASED_ABSOLUTE`, which is skipped during image relocation.

Backward Compatible Position-Independence

The notion of PIC is different for the PE format compared to ELF and Mach-O. Image relocation fixups can target read-only image segments even for PE shared libraries, which means that, unlike their counterparts in the other formats, most of their memory pages could not be shared if they were loaded at different load addresses across process address spaces [50]. Note that not sharing significant portions of shared library memory removes most of the benefits of the concept.

With the introduction of ASLR, the Windows kernel started pseudo-randomizing the load address of shared libraries and executable file [50]. Before the introduction of ASLR, this was the image base address and collisions caused an image relocation of the shared library and thus creating a copy. For this reason, image base addresss used to be carefully chosen to minimize the risk of collisions between process address spaces. To avoid requiring changes to the existing design of shared libraries, and to mitigate the COW penalty mentioned above, it still tries to load multiple instances of a shared library at the same load address across all processes [35]. To do this, it keeps track of the previous ASLR allocations and only draws the address from the available ranges. The image file is then fixed up when it is loaded from secondary memory, so that it appears to the rest of the existing shared library stack as if the image base address was the ASLR address, and it happened to be available. In case of a collision, the shared library still needs to be duplicated, but this is not a regression from the previous design. On the contrary, the pseudo-randomization reduces the risk of collisions compared to the previous manual optimization.

This approach has the obvious disadvantage that one process can learn information about another's address space via a side-channel. It also has to explicitly handle the case of an address collision, unlike ELF and Mach-O, potentially leading to increased memory consumption [35]. Support for PIC as implemented by the other formats has not been added to the PE specification or Microsoft Visual C++ (MSVC) [6].

2.1.4. Merging Read-Only Data with Executable Code

To reduce the impact of increased memory space requirements due to memory page boundary padding when tightening memory permissions, some build systems merge read-only data with executable code on the same memory page [5, 21]. This preserves immutability, meaning that the affected data cannot be corrupted to hijack the control flow. This is particularly important, because static memory allocations can affect computations, or the control flow directly, e.g. lookup tables, function tables, and so on.

At the same time, the more executable code there is, the more likely there is to be a suitable code-gadget naturally increases. Unfortunately, we have not been able to find any meaningful research on the impact of this.

2.2. Common Image Authentication Schemes

To support image authentication at a scale, several approaches have emerged for the kernel-space and user-space environments. In the following, we will look at the state of the art in operating system designs.

2.2.1. Linux Module Signature Format

ELF does not currently support digital signatures [4]. User-space applications and libraries are generally not digitally signed. To work around this limitation, the Linux kernel introduced the Linux Module Signature Format (see Figure 2.5) [10]. This is a trailer-based format independent of ELF that is appended to a Linux module to digitally sign it. The digital signature covers the whole image file (see Figure 1.9) [10].

Unfortunately, this introduces a slight ambiguity that is not trivial to resolve. Namely, the image file loader could process any Linux Module that has the Linux Module Signature Format file magic number at the end, e.g. as an unrelated datum at the end of its last image file section, as if it were digitally signed, even if it is not. To mitigate this, the Linux module needs to have some information about its file size, which for ELF files can only be obtained by parsing the image file section table [4]. While this is unlikely to cause any security or real-world usability problems in practice, it may be relevant for possible formalization efforts in the future.

2.2.2. Code Directory Hash and Image4 Signature Formats

Apple operating systems generally use the Code Directory Hash (cdhash) digital signature format for user-space applications and libraries [52, 53]. It stores a data structure with a cryptographic hash for each memory page of the Mach-O image, as this allows code authentication when loading them lazily during execution (see Figure 1.10).

For the boot process, including Secure Boot, Apple platforms use their proprietary Image4 format [8, 54]. Like the Linux Kernel Module Signature Format, it hashes the module as a whole. However, it is not just a simple

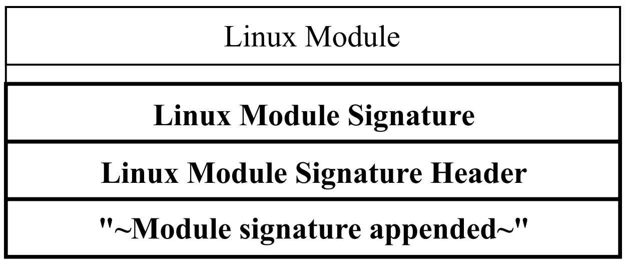

Figure 2.5.: *Linux Module Signature Format [51].*

The Linux Module Signature Format is trailer-based and independent of ELF. The trailer file magic number `"~Module signature appended~"` *identifies a Linux Module to which a digital signature has been appended. The file magic number is preceded by the digital signature header, which contains the information needed to process the digital signature, such as the cryptographic hash function and the digital signature size. Finally, the actual digital signature is located before the header.*

appended digital signature, but a sophisticated dictionary of various data stored in a separate file. For example, it may contain a unique machine identifier to allow personalized digital signatures.

2.2.3. Authenticode PE Signature Format

Authenticode is used by Microsoft for its operating systems, most notably Windows [55]. It is also the scheme adopted by the UEFI specification for Secure Boot, as there is no viable alternative for PE files. It is by far the most complicated image authentication scheme out of the four we cover (see Figure 2.6), while at the same time not integrating well with virtual memory. The hashing algorithm skips over some data in the image file header and its implementation for executable files used to tolerate trailing data [56]. Part of its complexity is to increase debugging flexibility [55], but it also significantly weakens its security properties. There have been several theoretical and practical attacks against this image authentication scheme, targeting both Windows and UEFI implementations [40, 56, 57, 58, 59].

2.3. UEFI Solutions and Reference Implementations

Now that we have looked at the state of the art in operating system designs, let us discuss the status quo and the solutions offered by the UEFI firmware ecosystem. In particular, this includes an attempt by the UEFI Forum to establish an image file format for the firmware space.

2.3.1. Terse Executable File Format

Many hardware designs have a very limited firmware storage space budget. For Intel architecture platform, the firmware often resides on a Serial Peripheral Interface (SPI) storage chip that may only provide a few tens of mebibytes of

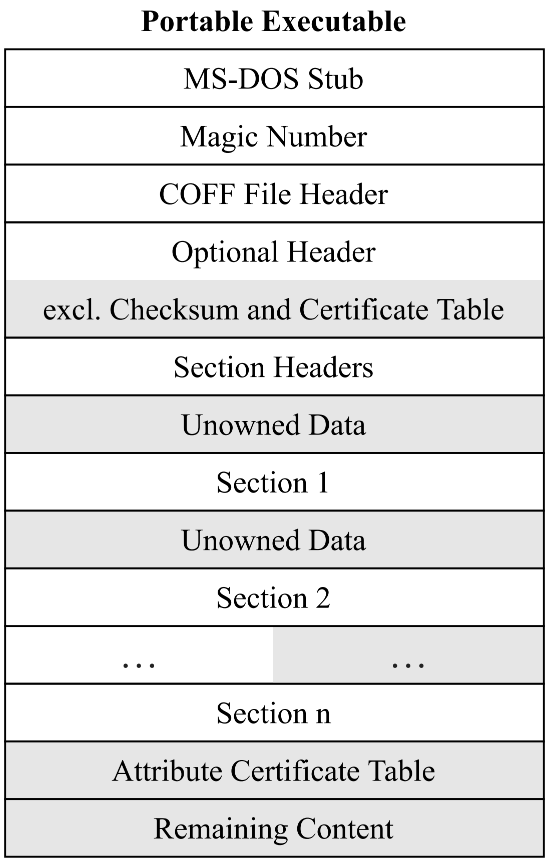

Figure 2.6.: *PE File Coverage by the Authenticode Scheme.*

The PE data covered by the Authenticode hash scheme. It cryptographically hashes data with a white background but not data with a grey background. 'Unowned Data' refers to data that is not covered by any PE image file section or data structure. In particular, any data following the Attribute Certificate Directory is not part of the PE Authenticode cryptographic hash [55].

space. The Terse Executable (TE) format is a curious approach to reducing the size of PE files [2]. Essentially, it replaces the multi-layer Portable Executable (PE) image file header with a single, denser image file header (see Figure 2.7). However, the conversion does not fix the internal offsets — this must be done by the image file loading [2, 21] — and it is unlikely to reduce the size of memory page-aligned execute-in-place (XIP) image files.

The simpler parsing properties of having only a single, simple image file header are contrasted by the complexity of considering the shift in data offsets and the semantic change to the image address space. For XIP files, the replacement of the PE image file header also shifts the addresses of all image segments [40]. To account for this, EDK II's *PeiCore* has a workaround to apply the shift to the load address, so that the image file header is technically misaligned, but the image segments are correctly aligned [21, 40]. This is especially important when enforcing memory permissions during PEI.

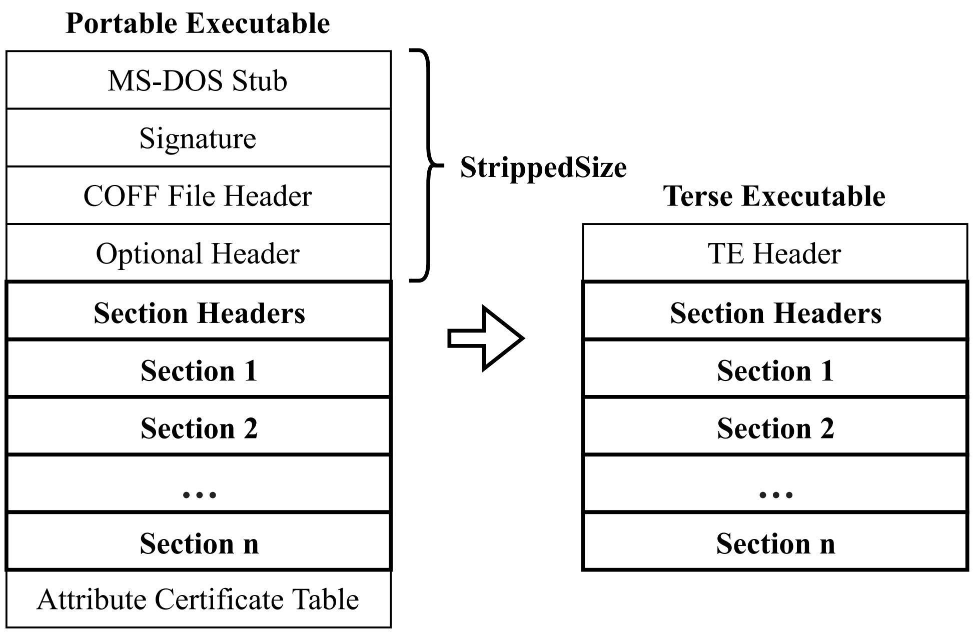

Figure 2.7.: *UEFI PI Terse Executable File Format.*

The UEFI PI specification defines the TE format in an attempt to reduce the size of image files compared to their PE counterparts. This is achieved by removing the PE image file header up to the PE image file section table and replacing it with a new TE image file header. The internal offsets remain unchanged and instead `StrippedSize` *is exposed to the image file loader so that it can consider the shift in data offsets.*

2.3.2. EFI Development Kit II

The reference implementation of UEFI PI and UEFI named EDK II is maintained by the TianoCore community under an open-source licence [21]. Most common UEFI implementations on the market fork and extend it, such as Microsoft Project Mu [60]. The Software Development Kit (SDK) includes most of the microarchitecture-independent components of the UEFI PI and UEFI specifications, as well as some ISA- and microarchitecture-specific industry code [21]. It also implements two fully-featured virtual QEMU [61] platforms, Open Virtual Machine Firmware (OVMF) for Intel architectures, and ArmVirtQemu for Arm architectures.

2.4. A Secure PE Loader based on Formal Methods

The current EDK II PE image file loader implementation has been subject to various security and reliability issues over the years [40]. Many problems are implementation details that could be fixed evolutionarily, but even the API itself is error-prone and unsafe. In response, we proposed a novel implementation with a new API that was designed using formal methods. The most critical correctness properties were formally proven using a Frama-C derivative, but late workarounds to support malformed PE files were not verified. Since its proposal, AUDK has completely replaced the old image file loading with this novel implementation and several parties have expressed interest in improving the security of this particular stack.

Despite its success in improving the security properties of the EDK II image file loader stack, the sheer complexity of the parsing logic and the various build-time configuration options to accommodate malformed real-world image files introduced a new kind of maintenance burden.

2.5. Image File Format Conversion

Compiler toolchains generally do not support all image file formats available. Also, they are often limited in platform availability. For example, the Apple Xcode toolchain supports only Mach-O files and is only available for Apple macOS, while MSVC supports only PE/COFF files and is only available on Windows. As such, it is not trivial to generate PE files on some platforms with their default compiler toolchains. To still support cross-platform and cross-format build environments, several tools to convert between image file formats have been developed.

2.5.1. ELF to PE: EDK II GenFw and iPXE elf2efi

For compiler toolchain configurations based on GCC and sometimes Clang, EDK II generates ELF files in the first step. To convert these into PE files, the TianoCore tool GenFw is used [21]. It converts ELF image file sections to PE image file sections while ignoring ELF image segments. This is due to the great flexibility offered by ELF image file sections by using custom static linking scripts to merge ELF image file sections as desired. It does not necessarily perform a strict translation but supports changing the relative offset between image segments to change the overall image data alignment requirement. GenFw also supports other operations, such as extracting Advanced Configuration and Power Interface (ACPI) tables from image files. An undocumented detail of the EDK II build system is that all image files are generated as XIP, regardless of their phase or purpose.

iPXE follows a similar approach with its separate elf2efi tool [62]. Unlike GenFw, its sole feature is a best-effort conversion of ELF to UEFI PE image files without any adjustments.

2.5.2. Mach-O to PE: Apple mtoc and Acidanthera ocmtoc

For Xcode-based compiler toolchains, EDK II generates Mach-O files in the first step. These are converted to PE files using the Apple tool mtoc, which is part of the Apple compiler toolset cctools [63]. Unlike TianoCore GenFw with ELF, mtoc translates the PE image file sections from the Mach-O image segments.

Unfortunately, mtoc has been subject to various bugs that have remained unfixed for years [64]. Recently, Apple released a version of the parent tool collection cctools which no longer contains mtoc. To the best of our knowledge, it is deprecated, at least for public use.

In response, ocmtoc emerged as a community fork [64]. It also fixed all known issues to date, including the generation of read-write-execute image segments and malformed debug directories. By default, Mach-O files generated by Xcode do not apply strict read-only memory permissions to immutable data, but merge them into the executable `__TEXT` image segment (see Section 2.1.4). In coordination with AUDK, ocmtoc implemented support for a strictly read-only image segment by moving the corresponding image file sections to a new `__DATA_CONST` image segment [49, 64]. Apple took a similar approach for the XNU kernel [8].

2.6. Improving and Testing for Memory Safety

The prevalence of memory safety violations and their exploitation has led to various research efforts to reduce both their frequency and their impact. The class of dynamic memory safety violations is often addressed by automatic memory management. This way, freeing memory is usually either handled by reference counting or garbage collection. In the firmware domain, this is currently not an option for performance and security reasons. However, in order to build confidence in the safety of our implementation, we have performed extensive safety testing, which we will now discuss.

2.6.1. Fuzz Testing

Fuzz testing is a technique to stress-test a fuzz target with randomly generated inputs. A typical goal is to observe safety violations based on a broad spectrum of inputs. In the context of memory-unsafe languages like C, this is particularly interesting for discovering memory safety violations (see Section 1.2.4). However, it is also possible to use it for functionality validation if there is a way to decide (with great confidence) whether the output is correct.

As most randomly-generated inputs are invalid and thus will likely be rejected quickly, fuzzers evolved to generate inputs in smart ways [65]. One of them is mutation-based fuzz testing, which lets you provide a corpus of valid inputs, which the fuzzer will mutate. This is commonly combined with coverage guidance, preferring mutations that increase some code coverage metric. A proven mutation-based and coverage-guided fuzzer is LLVM libFuzzer, which tightly integrates with the Clang compiler [66]. For coverage guidance, it uses SanitizerCoverage [67].

Beyond arbitrary mutations, sophisticated techniques like grammar fuzz testing surfaced [65]. It allows defining a grammar for inputs to generate valid or otherwise interesting inputs with a much higher probability. Of course, it can still be used to generate invalid inputs. With probabilistic grammar fuzz testing, the probability of getting different kinds of input can be influenced.

Finally, especially relevant for mutation-based fuzz testing, there are new approaches to combine it with symbolic execution to seed the corpus [68, 69].

2.6.2. Code Sanitizers

Code sanitizers consist of compiler instrumentation and sometimes shared libraries that can embed various checks to detect erroneous or suspicious behaviour. Two prominent examples are LLVM AddressSanitizer (ASan) and UndefinedBehaviorSanitizer (UBSan) [70, 71, 72]. The former can detect memory exploits and misaligned memory accesses, among other things. The latter detects undefined behaviour in the sense of C. Both can be combined with fuzz testing to perform basic empirical code safety testing. However, while this is a good way to increase confidence in the implementation, testing can never provide definitive guarantees. For memory-unsafe languages, the only way to derive them is through formal methods, which we will discuss next.

2.6.3. Formal Verification

Formal verification is a field of techniques for rigorously reasoning about the correctness of algorithms using mathematical methods. The most prominent tools in this area are proof assistants, such as Isabelle/HOL [73] or Coq [74]. While the former relies on classical logic, the latter is limited to intuitionistic logic, at least out of the box. Intuitionistic logic is the basis of the Curry-Howard Correspondence, which defines the isomorphisms commonly known as proofs-as-programs and propositions-as-types [75]. This allows programs to be generated from constructive mathematical proofs. One such product is the optimizing C compiler CompCert, which has been formally verified using Coq [76].

However, for various reasons such as code optimizations, tools have emerged to verify software written in traditional programming languages. One such tool is Frama-C [77], which is used in safety-critical industries, such as in aviation by Airbus [78]. It allows the verification of C programs for safety properties, such as memory safety, but also for functional properties, which can be provided as formal specifications using ANSI/ISO C Specification Language (ACSL) [79].

Formal methods are the only way to guarantee properties such as memory safety for languages without memory safety semantics. However, they are only used in safety- and security-critical scenarios because of their steep learning curve and the enormous effort (and thus cost) of formalizing and verifying the entire codebase. Sometimes verification covers only a critical subset of a software product. For Coq, the so-called kernel acts as a trusted core [74]. It is the only point of failure for the proof assistant. All tactics rely on the kernel to check their output, which are terms in a common proof language. If a tactic produces incorrect output, it may produce false negatives, but never false positives.

2.7. The Rust Programming Language

The Rust programming language (Rust) aims to be a low-level programming language with strong semantic guarantees from the compiler [15]. Its most interesting design detail is the ownership and borrowing model for memory allocation. It covers both call stack memory allocations and dynamic memory allocations and statically proves all memory accesses to be temporally safe. In addition, Rust dynamically validates memory bounds on access. This rules out all classes of memory safety violations and guarantees that the code is memory-safe. However, this model is limited and some memory-safe code cannot be expressed with it, such as doubly-linked lists. To still support such use cases, Rust supports unsafe code, which provides no more guarantees than other languages such as C. Another related safety feature is the detection of integer overflows, which can sometimes be abused to cause buffer overflows, or, more generally, out-of-bounds accesses.

2.7.1. The Borrow Checker and Temporal Memory Safety

Rust uses variable bindings, identified by a variable name, to reference allocated memory [15]. When the binding goes out of scope, it automatically frees the corresponding memory, much like automatic variables work in C. However, unlike C, the safety of this automatic memory management is statically proven. This requires that the binding is authoritative over the lifetime of the memory, i.e. the binding owns the memory. To allow other places in the code to still access the data, but not in a way that can introduce race conditions or dangling references, Rust introduced the concept of borrowing.

Borrowing can create both immutable and mutable references to memory. Code can create either an arbitrary number of immutable references or exactly one mutable reference. This works in the same way as a readers-writer lock and provides the same guarantees. Not only explicit reference creation takes advantage of this, but iterators also provide only immutable references, preventing iterator invalidation. Since references cannot semantically outlive their resource, use-after-free is completely mitigated. Finally, since references are only borrowed and never owned by two locations, there can be no reference cycles, which completely mitigates memory leaks.

There have been attempts to introduce Rust ownership and borrowing in C++. However, the researchers concluded that the C++ type system did not allow for this [80].

2.7.2. Rust in Linux

In 2021, a Request for Comments (RFC) was posted to the Linux kernel mailing list to discuss the addition of Rust as an official programming language for the Linux kernel [81]. However, unlike user-space applications, low-level code for firmware and operating system kernels has unique requirements. First, they may not provide a full-featured standard library for performance, efficiency, or security reasons. Second, critical interruptions of the control flow (e.g. exceptions and panics) are unacceptable, because there is no reasonable way to recover from them. This fundamentally deviates from the design of modern programming languages, including Rust. In order to meet the needs of the kernel, custom solutions and paradigms that are much closer to the traditional design of C programming have been discussed. Finally, in 2022, the initial Rust infrastructure landed in the Linux kernel tree [82].

2.7.3. Formal Verification and RustBelt

Because of its strict semantics, Rust can provide some guarantees at build-time that previously could only be established by formal verification (see Section 2.6.3). For five years, the Max Planck Institute for Software Systems (MPI-SWS) has researched the development of formal foundations for the Rust language as part of 'RustBelt', including compiler verification similar to CompCert [83]. Earlier this year, the official Rust types team was formed to pursue a formalization of the type system [84]. If successful, much of the burden of the formal verification of safety properties could be shifted from the individual project using Rust to the Rust compiler itself.

3. Requirements for a UEFI Executable File Format

In this chapter, we will perform requirements engineering for the proposal of a new executable file format for UEFI firmware. Section 3.1 provides insight into various techniques for improving memory security, some of which have been incorporated into the UEFI reference implementation EDK II. Then, Section 3.2 gives a broad overview of the related UEFI designs and technologies. Finally, Section 3.3 details the requirements of UEFI for image file formats and supporting code.

3.1. Memory Security

In response to the exploitation of memory safety violations, CPU architectures, operating systems, and firmware implementations have developed a variety of security approaches [27]. These do not address or fix the vulnerabilities themselves but aim to dramatically reduce their impact and the practicality of exploitation. We will now look at some of the more common techniques.

3.1.1. Memory Privileges

Virtual memory also allows the maintenance and isolation of multiple address spaces. This is particularly useful for isolating user-space processes and restricting access to the kernel-space address space via memory privilege. In general, there is one page table per user-space process, which is shared with the kernel-space. So, the kernel-space memory is part of the user-space address space, but not accessible to the corresponding user-space process due to memory privileges isolation. However, for side-channel-related security reasons [85, 86], some operating system configurations separate the page tables for kernel-space and user-space to map only the kernel-space memory required to bootstrap any system call (syscall). In the Linux kernel, this is implemented by a feature called page table isolation (PTI) [87]. To strengthen the isolation in the other direction, Intel introduced Supervisor Mode Execution Prevention (SMEP) and Supervisor Mode Access Prevention (SMAP), which restrict the execution of and memory accesses to user-space memory from kernel-space unless explicitly requested [11].

To address the performance penalty of the PTI approach and to simplify the implementation, Intel introduced Linear Address Space Separation (LASS) [88]. Unlike existing related solutions such as SMEP, SMAP, and PTI, LASS does not rely on page table information and therefore can be used without a page table walk. This is a new approach to mitigating timing-based side-channel attacks since deciding whether the address belongs to the kernel-space can easily be done in constant time.

3.1.2. Memory Permissions

Virtual memory management includes not only address space mappings and memory privileges, but also memory permissions. While memory privilege defines which CPU mode can perform operations in general, memory permissions define which operations can be performed. The typical operations that can be restricted by memory permissions are reading, writing, and executing memory. Most processors have restrictions on possible combinations [11]. However, historically unconventional memory permissions combinations such as execute-only are supported by some.

For security reasons, memory permissions should generally be configured as tightly as possible. In particular, a non-executable call stack is one of the basic defences against call stack smashing (see Section 1.4). However, it is important to note that page table changes can lead to performance degradation for many reasons, such as the need to flush translation lookaside buffer (TLB) caches. Also, memory permissions can only be applied on the level of memory pages, so tightening memory permissions may increase the overall memory requirements. Therefore, deciding the best configuration for each use case requires careful analysis.

3.1.3. Mutual Exclusion of Writing and Executing (W^X)

W^X is a paradigm in memory security that memory should never be writable and executable at the same time. This prevents so-called self-modifying code, which can be used to exploit vulnerable applications (often privileged) and bypass security measures such as digital signatures by modifying the code only after it has been authenticated (see Section 1.4). It also implies a non-executable call stack, as it is writable by nature.

However, there are also legitimate use cases for memory pages that are both writable and executable, such as just-in-time compilation (JIT). In response, there have been novel approaches to introducing W^X into JIT workloads. For example, Apple silicon provides efficient and thread-specific toggling between being writable and being executable, rather than having to modify the page table [89]. This avoids some of the penalties associated with typical memory permissions changes.

3.1.4. Control Flow Security

Another attempt to mitigate call stack overflows is the use of call stack canaries [27]. Typically, on entering the subroutine, it generates a pseudo-random value and stores it near the return address on the current call stack frame. Before returning, the subroutine compares the call stack canary with the original value from a safe location, usually a processor register, to see if it has changed. If the values do not match, a call stack overflow has occurred and the program terminates to prevent exploitation. However, an obvious shortcoming of this approach is that it does not catch indirect call stack corruptions that leave the call stack canary intact.

To address these shortcomings, sophisticated forward-edge CFI and shadow call stacks have been implemented by Central Processing Units (CPUs) [11, 12], operating systems [90], and UEFI firmware implementations [3, 21]. Forward-edge CFI is implemented in hardware and, when enabled, the compiler marks all jump targets with a special instruction. If the CPU has forward-edge CFI enabled, e.g. branch target identification (BTI) [12] or indirect branch tracking (IBT) [11], and a jump to an address without such an instruction occurs, the CPU throws an exception and the operating system may terminate the offending program. This severely limits the candidate pool for code-gadgets.

Shadow call stacks can be implemented either in software, e.g. Clang ShadowCallStack [91], or in hardware, e.g. Intel Control-flow Enforcement Technology (CET) [11]. Their purpose is to keep track of the history of the return addresses in a safe place so that the value in the call stack frame can be verified as authentic. Unlike call stack canaries, this also detects indirect call stack corruption.

3.1.5. Address Space Layout Randomization

Another technique to make code-gadgets harder to find is to randomize their locations [27, 50]. ASLR pseudo-randomizes the image base addresss of all shared libraries (called PIC) and executable files (called PIEs), the heap memory, and the call stack memory addresses. Unlike Windows [35], Linux and macOS even pseudo-randomize the image base address of shared libraries per process rather than globally.

As attacks against ASLR have surfaced, additional hardening, such as Function Granular KASLR that even randomizes function addresses within an image, was deployed [92].

3.1.6. Allocator Hardening against UAF Type Confusion

A relatively recent development in memory security is the hardening of dynamic memory allocation against use-after-free type confusion attacks. In this context, this does not refer to confusing the type of a live object, but between a previously freed object and a live object that now owns that memory. For its XNU kernel, Apple has implemented best-effort protection against confusing pointer and non-pointer data between reallocations [32]. This is achieved by type buckets, which are calculated at build-time. The build system chooses the buckets so that there is no overlap between pointers and non-pointer data across all its types. Once a virtual memory address has been allocated to a particular bucket, it can never be reallocated to an object from another bucket. This effectively prevents use-after-free vulnerabilities from being exploited to interpret arbitrary data as an address. While this does not mitigate the use-after-free vulnerabilities themselves, it can mitigate their exploitability [93].

3.1.7. Takeaways from Memory Safety and Memory Security

Memory exploits are not explicitly within the scope of this work. We will discuss ASLR and forward-edge CFI in the context of UEFI and image file next, but our contribution cannot help harden memory security. The key takeaway from exploring the field of memory security is that, while it is crucial to provide defence-in-depth against potentially stealthy and critical software exploits, it cannot give us sufficient guarantees, no matter how sophisticated it becomes. On the other hand, if we managed to guarantee memory safety, most techniques that harden memory exploits would not be required. As parsing executable files requires handling a lot of potentially malicious data, with the ultimate goal to execute it, memory safety is of utmost importance. Still, memory security should be employed or at least not explicitly hindered where possible.

3.1.8. Memory Security in Image File Formats and UEFI

The UEFI specification and EDK II include some security features discussed above [3, 21, 94]. Memory permissions, call stack canaries, and forward-edge CFI are supported by EDK II, while ASLR is still in the proof-of-concept phase [94]. Unlike operating systems, UEFI must guarantee that certain memory does not move across resets, which severely limits the potential of ASLR. While the most common examples are resuming from sleep states, EDK II supports memory type bins that are iteratively tuned in size [21]. The memory page and pool allocators will try to fit all dynamic memory allocations into these bins first, before resorting to other areas of memory. The design of this feature fundamentally contradicts the idea of ASLR, which also needs to be addressed in the future.

However, while EDK II theoretically supports tight memory permissions, some operating systems and OS loaders still assume data allocations to be executable or request read-write-execute memory permissions [95, 96]. Unfortunately, W^X is not yet widely supported due to implementation details that rely on shared memory between code and data [97]. For image file loading in particular, EDK II also does not support read-only memory, and instead silently makes it writable [98].

In the context of image files, forward-edge CFI only affects code generation, so it is independent of the image file format. However, each format requires some way of signalling that forward-edge CFI is supported by the image file [6].

3.2. The Unified Extensible Firmware Interface

Modern computer platforms require complex initialization to function as expected [2]. This can include tasks such as updating the CPU microcode, initializing the main memory, or booting the operating system. In the past, there were mostly handled by proprietary and closed firmware implementations, such as the Basic Input/Output System (BIOS) for Intel architecture CPUs [99]. With increasing complexity and the need for interoperability, the industry moved to a new modular and extensible alternative, known as the Unified Extensible Firmware Interface (UEFI) [3]. It has since become the supported boot specification for the Microsoft Windows operating system [7]. Companies such as Qualcomm also use it for various mobile device platforms, such as System-on-a-Chip (SoC) products used in Android phones and tablets [9].

Most UEFI implementations conform to both the UEFI Platform Initialization (UEFI PI) [2] and the UEFI [3] specifications. While the former deals with external compatibilities, such as providing services and specifications for applications and OS loaders, the latter deals with internal compatibilities, such as providing services useful for hardware initialization. Both are owned and published by the UEFI Forum.

The specifications specify the PE format for UEFI PI and UEFI modules of all kinds [2, 3]. SEC and PEI modules can alternatively be TE executable file. Our contribution is to replace this dependency with an optional alternative executable file format. To get an idea of the requirements for such an alternative, let us first discuss the basic workings of UEFI.

3.2.1. UEFI PI Phases of Execution

The UEFI PI specification describes several phases of execution (see Figure 3.1) [2]. This design follows the multi-phase process of hardware initialization used by different types of platforms. Some phases essentially extend UEFI concepts [2, 3]. For example, UEFI PI BDS implements the UEFI Boot Manager.

Security (SEC)

The Security (SEC) phase is the first phase to be invoked, for example by power events such as starting or rebooting the device [2]. In general, its purpose is to prepare the environment for the execution of generic C code. This may include updating the CPU microcode or checking the health of the CPU. The most important task however is to set up a temporary memory that can serve as a call stack. Traditionally, this is done using a technique called cache-as-RAM (CAR). Intel platforms implement this using the no-eviction mode (NEM), which disables cache line eviction on cache misses, thus guaranteeing the persistence of stored data. Due to its nature as the first software invoked, it serves as the Software Root of Trust (SRoT) of the UEFI PI firmware model. Once it completed all its initialization tasks, SEC invokes the PEI phase.

Pre-EFI Initialization (PEI)

The Pre-EFI Initialization (PEI) phase is the second phase to be invoked, usually immediately after the SEC phase [2]. In the past, its sole purpose was to initialize the permanent main memory in a simplified environment. However, with the introduction of features such as platform recovery (e.g. in case a firmware update fails), this is not strictly true with the current generation of UEFI PI implementations. Some modern platforms, such as the AMD Ryzen

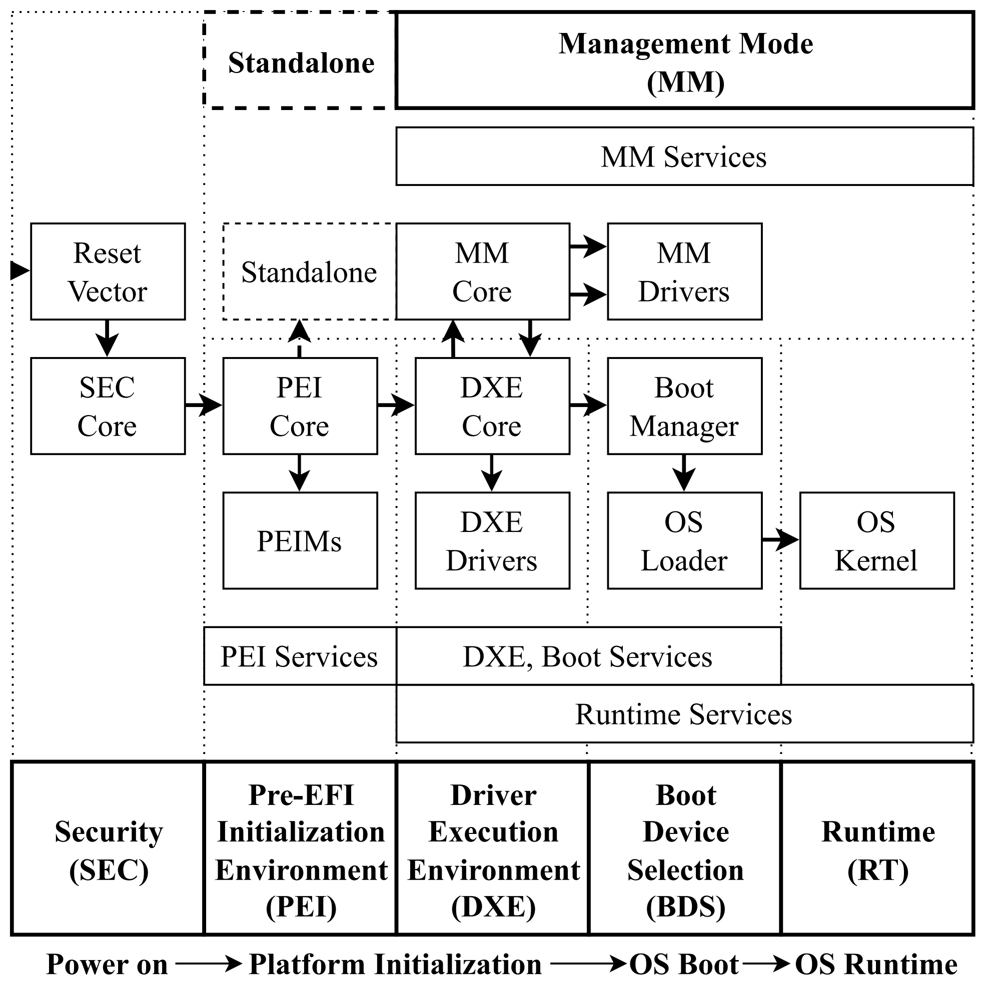

Figure 3.1.: *UEFI PI Phases of Execution.*

The UEFI PI phases of execution and the corresponding module dispatchers and services. For more information on the phases, see Section 3.2.1.

family of CPUs, no longer perform initialization of the main memory as part of the platform firmware and have instead moved it to less open hardware, such as the AMD Platform Security Processor (PSP) [100].

When resuming from platform sleep, PEI executes so-called S3 boot scripts to restore the platform state destroyed by the platform sleep state invocation [2]. These S3 boot scripts can be modified by PEI or subsequent phases to preserve their platform initialization operations. Consequently, UEFI PI skips subsequent phases when resuming from platform sleep.

Driver Execution Environment (DXE)

The Driver Execution Environment (DXE) phase is the third phase to be invoked [2]. However, it can be skipped for certain platform boot paths, such as resuming from platform sleep. Its primary purpose is to initialize the remaining hardware which was not yet initialized by the PEI phase and to establish platform security. Initialization can include the platform chipset, PCIe devices, storage interfaces, and more. DXE establishes platform security by locking down MM (e.g. no new MM communication buffers can be allocated after it has finished) and optionally write-protecting the firmware storage. It also implements Secure Boot and Measured Boot to authenticate Option ROMs and OS loaders (see Section 3.2.5).

Boot Device Search (BDS)

The Boot Device Search (BDS) phase is the fourth phase to be invoked [2], usually just after DXE. It may not be reached if the platform is reset early for any reason (e.g. when applying a firmware update). Its primary purpose is to locate and pass control to bootable code, such as an OS loader.

However, as many machines require a degree of customizability, it can also invoke built-in applications, such as a configuration utility or a boot menu for multiple operating systems, based on user interaction. To enable this, various peripherals such as mice, keyboards, video outputs, and the like (commonly referred to as human interface devices (HIDs)) need to be initialized. This is done by the UEFI Boot Manager, which is the abstract concept of BDS outside UEFI PI. The UEFI Driver Model allows drivers to probe devices and attach to them if they are supported. User input, such as keyboard hotkeys, can then trigger various actions, such as invoking the boot menu or rebooting the platform.

Transient System Load (TSL), Runtime (RT), and Afterlife (AL)

The Transient System Load (TSL) phase is more of a transition [2]. It begins with the invocation of the OS loader and ends when the operating system has finished transitioning UEFI to the Runtime (RT) phase, by providing kernel-space address mappings for the UEFI RT data. Unlike the other phases, RT does not control the system itself but is an aid to the operating system, which can call so-called UEFI Runtime Services on demand. The idea is to abstract hardware-dependent functionality with C code that can be called in the kernel-space context. The Afterlife (AL) phase is also not strictly a phase either, but the process of shutting down the platform.

Traditional and Standalone Management Mode (MM)

The UEFI PI Management Mode (MM) phase is not strictly a phase like the others, but once it is set up, it is entered on demand [2]. It can either be set up during the DXE phase and access all the services and UEFI protocols it provides, known as Traditional MM. Or it can be set up during the PEI phase and not access any external services on its own, known as Standalone MM. Derived from the Intel System Management Mode, its primary purpose was to handle power management functions during the operating system runtime. However, with modern Intel and AMD chips, this functionality has been moved to the CPU itself. MM remained as a high-privilege mode used for security isolation. For example, before loading third-party code, modern Intel and AMD platforms typically restrict write operations to the firmware SPI to MM, for security reasons. To still be able to write non-volatile RAM variables at runtime, a RT driver sends the request to a MM driver, which then writes the variable if allowed.

3.2.2. Module Dispatchers and the Image File Loader

UEFI PI module dispatchers handle the scheduling of modules in a correct order [2]. They process firmware volumes and repeatedly start all modules that have all their dependencies met. Dependencies, which can be PEIM-to-PEIM Interfaces (PPIs) or UEFI protocols, are described by the dependency expression (depex) section for UEFI PI PEI modules (PEIMs) and DXE drivers. As PPIs and UEFI protocols can be installed at any point in time, the loading order of the modules can only be determined at runtime. To load and start these modules, the module dispatchers uses a TE and PE image file loader. Strictly speaking, the module dispatchers themselves only process image files from the firmware storage, which are authentic if the firmware volumes are all digitally signed. However, the BDS phase may load UEFI drivers, applications, and OS loaders from untrusted storage, which pass through the very same infrastructure. These images are not subject to the same intrinsic trust and must be handled with great care to avoid exposing exploitable behaviour to attackers.

In addition to ensuring memory-safe parsing code, trust can be established by authenticating that the image file is trusted by a platform authority. Next, we will discuss Chains of Trust and how they can be used to authenticate the entire platform boot process.

3.2.3. Roots of Trust and Chains of Trust

Chains of Trust are the foundation of authority-based security designs. They start with an axiomatically trusted entity, the Root of Trust, which could be a product owner, for example. In user-space software designs, this can be implemented with a root digital certificate known to belong to the product owner, while in hardware designs it is typically a security chip that the product owner may have developed or validated. From there, the Root of Trust authenticates the next entity in the chain. Continuing with the hardware Chain of Trust, this could be its own firmware. Further down the line, there are multiple layers of components that may span across multiple supply chains and involve multiple product vendors. For example, the security chip firmware may authenticate a more general platform firmware, and the platform firmware may load a hardware driver provided by a third party. When the platform firmware authenticates this driver, it is represented as another link in the Chain of Trust. The general idea is that trust is transitive, so if the platform firmware trusts the driver, and the Root of Trust trusts the platform firmware, then the platform trusts the driver.

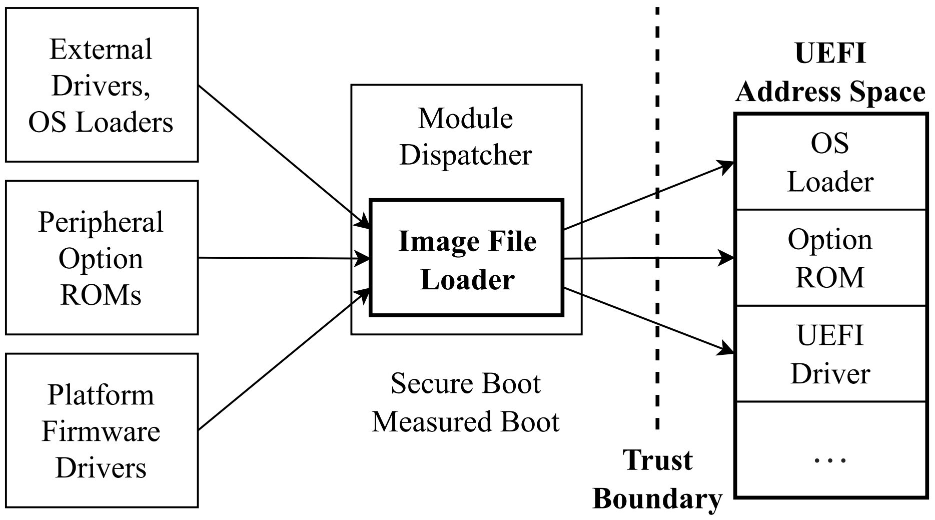

Figure 3.2.: *Role of the Image File Loader in UEFI PI and UEFI.*

The image file loader is a central component of the module dispatchers for PEI, DXE, and MM. While PEI and MM deal almost exclusively with platform drivers, DXE traditionally loads Option ROMs and OS loaders from untrusted media. As such, it must be resilient to memory safety violations caused by maliciously-created image files. The same applies to image authentication, as all image run with critical privileges.

However, trust is not necessarily a binary property. For example, an entity may only be trusted only with certain platform resources and privileges. Transitive trust must never extend the scope of trust. For example, certification authorities can never sign digital certificates or data outside their scope. By limiting their privileges, the impact of abuse and exploitation is limited, which is why links towards the root are privileged and well protected (e.g. protected by hardware security modules (HSMs) and physical security), while links towards the leaf are a lot more limited but accessible.

To reduce the risk of unnoticed exploitation, digital certificates have a limited validity timeframe and must be renewed before expiring. Leaf digital certificates are usually only valid for a few months or around a year, while a root digital certificate may be valid for multiple decades. The higher the chain a link is, the harder it is to renew. In the event of a theoretical security vulnerability or exploitation, one can revoke trust in any link in the chain. This is also not necessarily binary — optionally, with a timestamp of revocation, data signed prior to revocation can still be trusted.

Besides a plain Chain of Trust, digital signatures and trusted timestamps can be used to implement more sophisticated security features. For example, with personalized (in the sense of the target computer) and on-demand signing (which enables the use of nonces for an interactive update process), soft downgrade protections can be implemented. With a naive approach, after a configurable amount of time, the update server simply stops signing old updates and their digital signatures expire. Apple implements a more sophisticated approach that roughly follows the same ideas [101].

Examples of Hardware Roots of Trust are Intel Management Engine (Intel ME) and Apple T2 [101, 102]. Next, we will discuss how Verified Boot uses them to authenticate the entire platform firmware using a Chain of Trust.

3.2.4. Verified Boot and Intel Boot Guard

Verified Boot applies establishes a Chain of Trust for the platform firmware to authenticate the entire boot process [102]. Intel Boot Guard is one of the technologies that make this possible (see Figure 3.3). With it, Intel ME acts as a Hardware Root of Trust (HRoT), which verifies the Initial Boot Block (IBB) of the platform firmware. As the reset vector resides on secondary SPI memory and Intel platforms still require a lot of code to initialize the main memory, the said IBB generally covers all pre-memory firmware volumes. To mitigate TOC/TOU attacks when shadowing the post-memory firmware volumes into the main memory, they are covered by a separate OEM Boot Block (OBB), which in practice is more of a logical collection of firmware volumes. The firmware exposes authenticated cryptographic hashs for them, which allows their authentication in software by PEI after they have been shadowed (EDK II *FvReportPei*). Due to the generally low speed of SPI memory, authenticating after shadowing improves the boot performance.

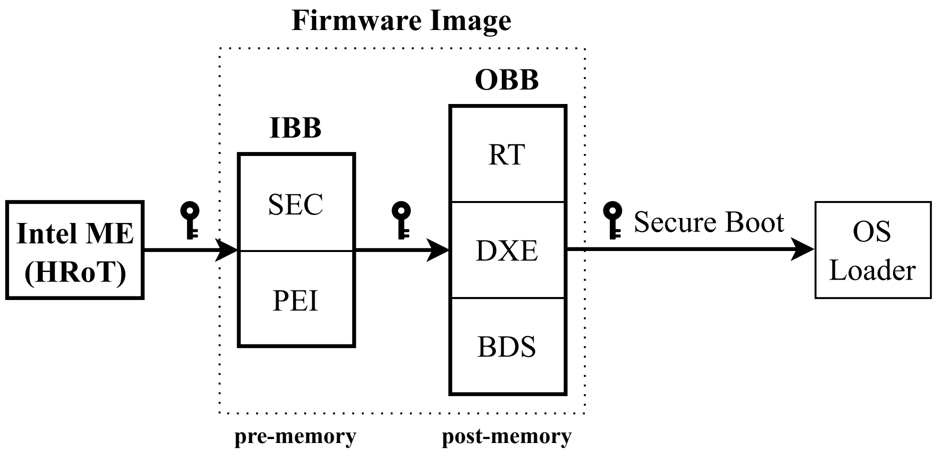

Figure 3.3.: *Verified Boot based Intel Boot Guard.*

Intel Boot Guard uses Intel ME as the HRoT for platform firmware authentication. It authenticates the IBB, which contains all the code required to run before initializing the system memory. The IBB is then responsible for authenticating the OBB once it has been shadowed into system memory. If important data resides outside both blocks or if anything from the OBB is authenticated before shadowing, the scheme may not be secure.

Apple moved the entire firmware image and authentication logic to their HRoT Apple T2 (see Figure 3.4). This will mitigate all the issues discussed above.

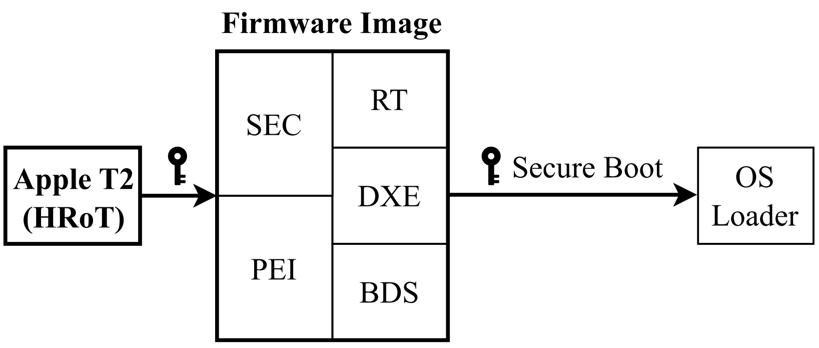

Figure 3.4.: *Verified Boot based on Apple T2.*

Similar to Figure 3.3. However, unlike Intel ME, Apple T2 authenticates the entire firmware image as a whole before powering on the main processor. This means that there can be no data that is not authenticated. Since the HRoT itself carries the firmware image, there can be no TOC/TOU attacks related to shadowing to system memory.

3.2.5. Secure Boot and Measured Boot

The UEFI specification has not only advanced the operating system boot process and the maintainability of firmware implementations, but it also introduced several new security features.

One of them is Secure Boot, an umbrella term for a software stack in UEFI firmware implementations to cryptographically authenticate image files using digital signatures. In the consumer market, this means that devices are generally shipping with the 'Microsoft Windows Production CA 2011' certification authority pre-installed [103]. Microsoft also provides the 'Microsoft 3rd Party UEFI CA' certificate authority to third parties to centralize the Secure Boot management, e.g. for the Linux operating systems landscape [104]. The operating systems by Microsoft and Apple expand code authentication into the user-space. While Windows only authenticates executable files and shared libraries, those by Apple generally authenticate the entire system image [101]. Recently, a flexible framework to authenticate (a subset of) the system image independent of the operating system has been showcased [105]. Most operating systems based on Linux do not have any load-time authentication at all, but the kernel supports dm-verity and fs-verity, which are leveraged by Android and ChromeOS [10].

Another security feature specified by the UEFI specification is Measured Boot, a currently enterprise-oriented scheme to collect critical information about the boot environment and send it to a remote attestation entity, usually a server. This information may include hardware and firmware configuration or loaded drivers, depending on the platform policy.

3.2.6. Human Interface Infrastructure (HII)

UEFI drivers and applications may expose and consume UEFI Human Interface Infrastructure (UEFI HII) concepts. They are the foundation of user input and output in UEFI, such as fonts, localization, and graphical interfaces as a whole. The UEFI HII database aggregates resources from so-called UEFI HII packages. Conventionally, the PE resource directory carries a UEFI HII package list that encodes these. It is then located by the module dispatcher at load-time and installed as an UEFI protocol on the module's UEFI handle [3]. EDK II however also supports exposing UEFI HII package data to the module via C variables, depending on the `UEFI_HII_RESOURCE_SECTION` build option [106, 21].

Outside of modules consuming their own UEFI HII packages, the EDK II Shell utilizes the separate PE resource section to extract help strings. This requires the PE file to be fully loaded, including setting up permissions to execute. Alternatively, the EDK II Shell supports bundled help strings and manual files [21]. The overcomplicated parsing logic of the PE resource directory has been subject to functional defects [40].

3.3. UEFI Requirements for Executable Files

Because UEFI differs from the design of modern operating systems in several ways, it has unusual requirements and use cases for image files. We will take a look at the most important unique scenarios below.

3.3.1. Execute-in-Place, Preloading, and Shadowing

Some platforms do not have any main memory available upon platform reset and instead repurpose dramatically smaller cache memory in order to support things like call stack memory, as required for the execution of C code [2, 14]. Firmware developers generally call this CAR. As said cache memory is sparse, Intel platforms running UEFI firmware implementations utilize a technique called XIP, which lets the CPU fetch and execute instructions directly from the firmware storage (generally connected via SPI) [2]. For this, the CPU maps the firmware storage at an address known at build-time [11]. To prevent accidental modification of the firmware storage, the MMIO mapping is generally read-only. Thus, XIP images cannot write to global variables EDK II.

The build tools store XIP images 'preloaded' so that they can be executed without loading or relocating at run-time [2, 21]. Image preloading in this context refers to performing image file loading and image relocation to the predetermined image base address at build-time. This especially implies that the firmware storage must hold the entire image address space without compression techniques, such as omitting trailing zero values of image segments as done by image files.

When main memory has been initialized and is available, important PEIMs can be transferred from CAR memory to the main memory by a technique called PEIM shadowing [2]. This process copies the entire image address space from the old to the new memory location and performs image relocation. As the same applies to call stack and heap memory, the PEI services provide a procedure for pointer conversion every PEIM must call on all pointers to such memory.

3.3.2. Runtime Image Relocation

Runtime image relocation, similar to PEIM shadowing (see Section 3.3.1), is used for RT drivers to support virtual memory mappings decided by the operating system [2]. In macOS, this has the side effect of extending KASLR to the UEFI runtime memory, as it is mapped adjacent to the operating system kernel address space [8].

By the time the firmware performs runtime image relocation, likely all the RT drivers have already been executed. This means that the targets of image relocation fixups may have changed, e.g. by changing global variables that were statically initialized with pointers. There are three ways to handle this situation tolerantly:

1. Apply the image relocation fixups as usual. This can cause faults if overwriting global variables that were statically initialized with an in-image address with an external address (see Source Code 3.1). The UEFI specification does not explicitly describe this as the standardized method, but there is no implication that there should be any workaround [3].

2. Apply an image relocation fixup only if its value is unchanged. This can cause bugs when overwriting global variables that were statically initialized with an in-image address with a different in-image address and not calling `ConvertPointer()` (see Source Code 3.2). EDK II and real-world firmware took this approach [21].

3. Apply an image relocation fixup only if its value is in the image address space. This would work correctly for most cases. However, at least Clang can emit image relocation fixup values that are not addresses themselves, but offsets to constants that form an address. While we have only been able to observe this for a single driver (EDK II's *FatPei* [21]) and only for the Intel 32-bit architecture, this scenario would cause an extremely rare and difficult-to-debug fault. We are not aware of any implementations of this behaviour and have only described it for completeness.

As we can see, all tolerant approaches can lead to unobvious faults and bugs. Intolerant handling, i.e. aborting if an image relocation fixup value does not match its old value, does not suffer from any of these problems. However, it does limit the expressiveness of RT driver code, since this constraint may reject valid C code.

3.3.3. Mixed-Mode Execution

Firmware implementations for 64-bit CPUs based on Intel architecture typically implement mixed-mode execution [107]. The CPU powers up in 16-bit mode [11]

```c
uint8_t mRuntimeDatum = 0;
// This emits an image relocation fixup.
char    *mRuntimePtr  = &mRuntimeDatum;

// Change the address of mRuntimePtr to something
// outside the current image before runtime
// relocation.
void PreRuntimeReloc() {
  mRuntimePtr = NULL;
}

// Naive operation performed by the firmware on
// runtime image relocation.
// void AtRuntimeReloc(uintptr_t ImageSlide) {
//   mRuntimePtr += ImageSlide;
// }

// Process mRuntimePtr after runtime image
// relocation.
void PostRuntimeReloc() {
  if (mRuntimePtr != NULL) {
    *mRuntimePtr = 0;
  }
}
```

Source Code 3.1.: *Fault from Naive UEFI Runtime Image Relocation.*

*Faulting program assuming naive UEFI runtime image relocation. This strictly assumes the UEFI specification's description of `SetVirtualAddressMap()` [3]. As `mRuntimePtr` is initialized with the absolute address of a global variable, an image relocation fixup is emitted for it (see Section 1.5.5). Before entering the runtime phase, we set the pointer to **`NULL`**. Due to the recorded image relocation fixup for the previous address, **`NULL`** is relocated, and the final value is exactly `ImageSlide = NewBase - OldBase`, i.e. a rogue pointer. After the runtime image relocation has succeeded, we run some code that changes the value of `mRuntimePtr` if and only if it is not `NULL`, a common way to signal pointer liveness. At this point, we corrupt arbitrary memory, even though the code is semantically correct.*

```
uint8_t mRuntimeDatum1 = 0;
uint8_t mRuntimeDatum2 = 0;
// This emits an image relocation fixup.
char    *mRuntimePtr   = &mRuntimeDatum1;

// Change the address of mRuntimePtr to something
// else inside the current image before runtime
// relocation.
void PreRuntimeReloc() {
  mRuntimePtr = &mRuntimeDatum2;
}

// Real-world operation performed by the firmware
// on runtime image relocation.
// void AtRuntimeReloc(
//   void      *OrigPtr,
//   uintptr_t ImageSlide
//   ) {
//   if (mRuntimePtr == OrigPtr) {
//     mRuntimePtr += ImageSlide;
//   }
// }

// Process mRuntimePtr after runtime image
// relocation.
void PostRuntimeReloc() {
  mRuntimeDatum2 = *mRuntimePtr;
}
```

Source Code 3.2.: *Bug from Real-World UEFI Runtime Image Relocation.*

Erroneous program assuming real-world UEFI runtime image relocation. This bug mirrors the fault from Source Code 3.1. Real-world UEFI implementations only relocate values that remain unchanged [21], which mitigates the previous fault. Before entering the runtime phase, we change `mRuntimePtr` to point to `mRuntimeDatum2`. As its address is different from that of `mRuntimeDatum1`, the image relocation fixup is not applied, even though the address is within the image address space. After the runtime image relocation has succeeded, we run some code that should effectively assign `mRuntimeDatum2` to itself. However, due to the incorrect address of `mRuntimePtr`, we are reading arbitrary memory. One could argue that according to the UEFI specification, `ConvertPointer()` should have been called on `mRuntimePtr`, which resolves the bug [3]. However, the vague wording about reapplying of image relocation fixups and the correct behaviour for unmodified pointers make the situation ambiguous.

and SEC switches it into 32-bit mode, which is the main operating mode throughout PEI [21]. Only when transitioning into DXE, the CPU finally switches into 64-bit mode. One of the main reasons for mixed-mode execution is the reduced memory requirements of the 32-bit code [108, 109]. While work is underway to move PEI completely to 64-bit mode [108, 109] and future microarchitecture proposals deprecate 16-bit and 64-bit modes for low-level operation [110], mixed-mode execution is a strict requirement for the time being. Most importantly, it means that a single image file can carry code targeting multiple ISAs, which is particularly relevant for image relocation.

3.3.4. Fixed-Address Loading

In stark contrast to ASLR, some embedded system designs mandate fixed memory maps. This is a typical consequence of prohibiting the use of dynamic memory allocation to establish worst-case guarantees for memory usage and allocation times (see Section 1.2.3). To achieve this, the build system must, among other things, statically allocate memory for all images, much like the compiler statically allocates memory for global variables (see Section 1.2.1). As such, all images can be pre-linked, moving image relocation to the build-time. As a side effect, this saves space, because the image relocation table can be stripped.

EDK II implements this concept opportunistically [21]. Image files can report that they were compiled with fixed-address loading enabled and the module dispatcher will attempt to allocate it at its fixed load address. To do this, the module dispatcher reserves a memory region and manages the memory page allocation with a bitmap. If there is an allocation collision, the image file is loaded at a different address. UEFI does not support disabling dynamic memory allocation and some core interfaces require its usage [2, 3].

As an alternative to traditional fixed-address loading, EDK II supports a fixed-offset mode, which works the same way but fixes an address relative to a base address chosen by the firmware. This is the only mode available for the MM phase. With this mode, the image relocation fixups cannot be stripped.

4. Designing a Secure and Space-Efficient Executable File Format

This chapter covers some security, efficiency, and portability aspects of designing an executable file format for UEFI. First, in Section 4.1, we will introduce a central component of our work, an abstraction for UEFI images. In Section 4.2, we will provide the scope of UEFI image files formats. Finally, Sections 4.3 to 4.5 deal with the ins and outs of parsing and encoding regarding performance, space efficiency, and security.

4.1. Defining a UEFI Image Abstraction

In this section, we will define an abstraction for UEFI images. Although it is mostly a natural consequence of the similarities between the various image files formats and the unique requirements of UEFI firmware, it is one of the most important foundations of our work, and will be used in several places to design the high-level architectures.

4.1.1. Data Structures of UEFI Image Files

Several data structures are common to image files formats. In particular, we will look at those relevant to a UEFI use-case. Some metadata are not universal concepts, such as the subsystem identifier, but they are introduced by the EDK II build system [4, 5, 6, 21]. We have identified the following data structures as relevant:

- **Image Metadata:**
  - **Machine Identifier:** Specifies the host ISA supported by the image file.
  - **Subsystem:** Specifies the type of the image (e.g. boot-time driver, runtime driver, or application).
  - **Image Relocation Fixups Stripped:** Whether image relocation fixups have been stripped. This is necessary to distinguish between whether the image does not require image relocation fixups, or whether they have been actively removed.
  - **Base Address:** The address of the image address space from which the loaded image can be executed without image relocation.

  - **Entry Point Image Offset:** The offset of the image entry point within the image address space.

- **Segment Table:** A table of image segments:
  - **Name:** An arbitrary name that identifies the image segment.
  - **Image Offset:** The offset within the image address space to which the image file loader loads the image segment data.
  - **Image Size:** The size, in bytes, of the image segment data within the image address space.
  - **File Offset:** The offset within the image file from which the image file loader loads the image segment data.
  - **File Size:** The size, in bytes, of the image segment data within the image file.
  - **Permissions:** The memory permissions that may restrict whether the image segment data within the image address space can be read, written, or executed.

- **Relocation Table:** A table of image relocation fixups:
  - **Type:** Specifies the type of image relocation fixup to apply. This specifies the format of the data read or written at the image relocation fixup offset.
  - **Image Offset:** The offset within the image address space the image relocation fixup targets.

- **Additional Information:**
  - **Debug File Path:** The path to the file with the image symbols on the build machine.
  - **Signature Table:** An embedded table of one or more digital signatures, which may cover the image file as a whole or the image segments individually.

4.1.2. Operations on UEFI Image Files and Images

Similarly, common operations can be applied to image files and image. The concept of runtime image relocation is introduced by UEFI, no image file supports a similar concept [3, 4, 5, 6]. We have identified the following operations as relevant:

- **Hash:** Compute a cryptographic hash for an image or image file. To be secure, it must cover all image segments' data and anything that might change them, such as the image relocation table.

- **Load:** Set up the image address space from an image file.

- **Relocate:** Change all internal absolute references to move the image to a new address.

- **Runtime Relocate:** Same as Relocate, but happens at runtime, after code has possibly been executed.

4.2. Scope of a UEFI Image File Format

Unlike the previously discussed image file formats, the proposed UEFI Executable (UE) image file format (see Appendix B) design is explicitly limited in functionality and scope:

1. No support for static linking. The build system generates UE files from files of the ELF, Mach-O, and PE formats, which support static linking [4, 5, 6]. Only the final image files is converted to UE, which hence does not need to support any static linking features.
2. No support for dynamic linking. Image symbol resolution is a complex process in modern image file formats and is often implemented with various optimizations to ensure high performance. UEFI PI and UEFI do not currently support dynamic linking and instead rely on PPIs and UEFI protocols (shared data structures indexed by a database) [2, 3]. The proposed alternatives are static linking to combine modules, or to use pre-linking, i.e. to perform dynamic linking at build-time, resulting in a composite module (see Section 8.3.2).
3. No considerations for image file loading supported by virtual memory management. UE is strictly for use with firmware at boot-time. UEFI has no concept of virtual memory management at the time of writing and its design is far from ready to support such a subsystem. As such, the image file loading semantics will be strictly based on explicit copying and features such as RELRO are not applicable.

As such, we propose UE strictly as an image file format. For the image file data structures, this means the following:

- No concept of image file sections (see Section 1.5.3).
- Only one, centralized image relocation table (i.e. no concept of image relocation fixups per image file section).
- No concept of image symbol tables.

4.3. Memory Layout, Binary Portability, and Parsing Efficiency

The C programming language is the basis for various low-level software, such as operating systems, drivers, and also the entire UEFI specification and firmware stacks based on EDK II [3, 10, 21]. Its heavy reliance on pointers allows binary data, such as the contents of a binary file, to be immediately interpreted as structs [14]. This can be used to parse such data in place, which allows for avoiding decoding operations and data copying. Since pointer dereferencing usually translates directly into memory instructions, the pointers must satisfy the processor's constraints on memory accesses. Therefore, each C data type has a data alignment requirement, which ensures that the datum is at an address that is a multiple of said data alignment requirement.

4.3.1. Natural Data Alignment and Data Structures

For basic data types, their data alignment requirement is often equal to their size requirement, which is known as natural data alignment. The UEFI PI and UEFI specifications declare natural data alignment requirements for all data types [2, 3], so it is safe to assume. In general, the widely-used Intel architecture ABIs require at most a natural data alignment for all basic data types [111]. The same is true for the Arm ABIs [112, 113]. Due to the fragmented and volatile nature of ISAs, it is not possible to cover them all, but we have not found any counter-examples. Note, however, that for some ABIs, the data alignment requirements do not exceed the target machine word size [45, 112]. Depending on the ISA, natural data alignment can allow for more efficient code generation [114]. This leads to:

Requirement 1. *The data alignment requirement of all basic data types must be treated as their natural data alignment requirement, regardless of the ABI data alignment requirements.*

For aggregates (structs and arrays) and union types, the data alignment requirement is usually the largest data alignment requirement among its members. Since data alignment requirements are practically always powers of two (due to the binary nature of processors and because C requires them to be a power [14]), this guarantees that the data alignment requirements of all members are met. Structs may require a special measure called data structure padding, which means inserting bytes of an unspecified value to fill a gap to the next data alignment boundary (see Source Code 4.1) [14]. Since, as we have seen, data alignment requirements can differ between ISAs, obviously not all structs are universally binary-compatible across ABIs. This leads to:

Requirement 2. *Each structure must be composed in such a way that no data structure padding is required if the natural data alignment requirements were met for the basic data types.*

4.3.2. Implications of Data Structure Packing

To allow avoiding the space overhead of data structure padding, many of the common compilers support data structure packing, e.g. through the MSVC extension `#pragma pack` [115]. This can be used to instruct the compiler to limit the data alignment requirement of struct and union types. When forcing the data alignment requirement to 1 byte, a data type is byte-packed. For structs, this effectively prevents the insertion of data structure padding (see Source Code 4.2 and fig. 4.1). The compiler then guarantees that all direct

```
// sizeof = 12, _Alignof = 4
struct {
  // offsetof = 0, sizeof = _Alignof = 1
  uint8_t  a;
 [// offsetof = 1, sizeof = 3, _Alignof = 1]
 [uint8_t  __pad1[3];                      ]
  // offsetof = 4, sizeof = _Alignof = 4
  uint32_t b;
  // offsetof = 8, sizeof = _Alignof = 1
  uint8_t  c;
 [// offsetof = 9, sizeof = 3, _Alignof = 1]
 [uint8_t  __pad2[3];                      ]
};
```

Source Code 4.1.: *C Structure Padding.*

The explicitly declared members of the structure are a *and* b*. The structure is not byte-packed, and we assume natural data alignment requirements hold. As* b *must have its data alignment requirement of* 8 *bytes met, the compiler inserts data structure padding preceding this structure member, here illustrated as the undeclared structure member* p *(not technically accurate). This data structure padding is not directly accessible, and its content is undefined at all times.*

accesses to struct and union members are safe, regardless of the address of the accessed member. However, if it declares members with types that are not byte-packed, creating and accessing pointers to them qualifies as undefined behaviour [14].

4.3.3. Basic Data Type Memory Layouts

Having looked at the layout of members within a struct, another thing to consider is the layout of the basic data types themselves. Endianness describes the byte order of a multi-byte basic data type. The two typical schemes are big-endian and little-endian. Take the 32-bit value $\mathrm{AABBCCDD}_{16}$ as an example. Its big-endian encoding corresponds to the following byte layout in memory:

```
AA BB CC DD.
```

On the other hand, the little-endian layout is:

```
DD CC BB AA.
```

While a detailed discussion of endianness and the technical reasons for it is beyond the scope of this thesis there is an obvious design advantage to each of the two schemes just provided: With big-endian, low-level debugging and the like is easier because the byte order follows the natural reading layout. For little-endian, value truncation may not require explicit instructions compared to big-endian:

```c
#pragma pack(1)
// sizeof = 6, _Alignof = 1
struct {
  // offsetof = 0, sizeof = 1, _Alignof = 1
  uint8_t  a;
  // offsetof = 1, sizeof = 4, _Alignof = 1
  uint32_t b;
  // offsetof = 5, sizeof = 1, _Alignof = 1
  uint8_t  c;
};
#pragma pack()
```

Source Code 4.2.: *C Struct Byte-Packing.*

The structure is byte-packed, which extends to all of its members, and thus everything has a data alignment requirement of 1 byte. Data structure padding never appears in byte-packed structs.

Take the example from above, i.e. the 32-bit value $AABBCCDD_{16}$. Now, we want to truncate it to a 16-bit value, leaving us with $CCDD_{16}$. We currently have a pointer to the 32-bit value. For big-endian, we can see that the 16-bit value

```
CC DD
```

is at an offset of 2 bytes relative to the 32-bit value. With little-endian, the value

```
DD CC
```

is at an offset of 0 bytes relative to the 32-bit value. Thus, little-endian saves one arithmetic operation when retrieving the address of the truncated 16-bit value compared to big-endian. Today, most general-purpose CPUs use the little-endian layout, and UEFI currently mandates it [3, 11, 12]. This leads to:

Requirement 3. *All basic types must be stored in the little-endian layout.*

4.3.4. Signed Integer Encoding

The two's complement is the standard choice for encoding signed integers for modern technologies. It is also the de facto standard for common general-purpose ISAs [11, 12]. Also, recent revisions of the C standard guarantee it as the only representation for its specified-width integer types [14].

Mathematically speaking, it is the positive complement in $\pmod 2)^N$. It is well known that it holds that:

$$\forall n, N \in \mathbb{N}_0. \quad -n \equiv 2^N - n \pmod{2^N}$$

Proof. $-n \equiv 2^N - n \Leftrightarrow 0 \equiv 2^N \Leftrightarrow 0 \equiv 0 \pmod{2^N}$ □

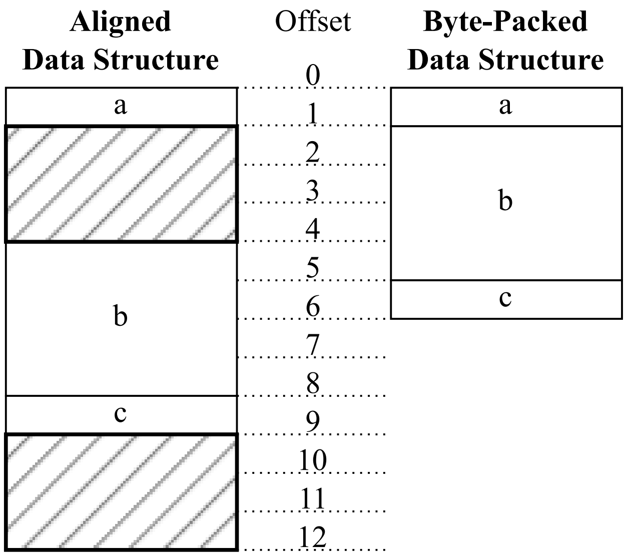

Figure 4.1.: *Struct Byte-Packing.*

N-bit integer arithmetic computes results mod 2^N for addition, subtraction, and multiplication. The division operation is the regular integer division. In technical terms, the reduction mod 2^N is commonly called an integer wrap-around or integer overflow. Following the proposition above, it is easy to see that the same operator definitions apply to the two's complement.

For any positive number, one can easily compute the negative number of the same value by inverting all the bits and adding 1. For example, for -1 and $N = 8$, we have:

$$\begin{aligned} 1_{10} &= 00000001_2, \\ -1_{10} &= 11111111_{2C}, \\ 11111111_2 &= (2^8 - 1)_{10}. \end{aligned}$$

Historical alternatives to the two's complement were the ones' complement and sign and magnitude [14]. Both include an explicit definition for -0 and arithmetic operations are not trivial to implement, e.g. ones' complement addition requires an end-around-carry [116]. While other representations exist, they are not widely supported by general-purpose CPUs and not supported by UEFI. This leads to:

Requirement 4. *All signed integers must be encoded as two's complement.*

4.3.5. Floating-Point Encoding

For floating-point numbers, Institute of Electrical and Electronics Engineers (IEEE) 754 is a well-established standard for both representation and computation rules [117]. While some hardware does not strictly follow the computational rules, and compilers them to be partially ignored for performance and efficiency reasons, the representation is de facto universal [34]. In fact, the C standard provides Annex F as a mapping of C floating-point types to IEEE-754 types, but not to any other encoding [14]. This leads to:

Requirement 5. *All floating-point types must be encoded according to the Institute of Electrical and Electronics Engineers (IEEE) 754 standard.*

4.3.6. Character and String Encodings

The current standard for character encoding today is Unicode. It specifies several encoding formats, such as UTF-8 and UTF-16. The former is a strict superset of ASCII, the traditional 7-bit standard for control characters and the English alphabet, and is also widely used across different technologies [118]. UEFI mostly uses UCS-2 encoding, which is a strict subset of UTF-16 [3, 118]. For UCS-2, all characters occupy exactly 2 bytes. This allows many non-English characters to be represented but avoids the flexible-length parsing logic required by UTF-8. While UCS-2 is binary-incompatible with ASCII, all UCS-2 characters representable by ASCII can be converted to it by truncating to 7 or 8 bits (bit 7 is always 0 for characters representable by ASCII).

Besides Unicode, every recent alternative character set we have come across also extends ASCII. Since localization is not an issue for a binary file format, ASCII is sufficient as a de-facto universal standard character encoding.

For string encoding, we opt for zero-terminated ASCII strings. Since UEFI primarily targets C, this is essential to harden the design against memory safety violations. Despite the availability of safe string functions that take an explicit size argument, EDK II in particular often relies on zero-termination [21]. This leads to:

Requirement 6. *All characters must be encoded in the ASCII format and all strings must be zero-terminated.*

4.4. Bit-Packing, Backward Compatibility, and Space Efficiency

Space efficiency is one of the areas we are gradually improving over the existing TE format. To achieve this without sacrificing the flexibility of the image file format, we will now discuss various ways of compressing data while remaining portable and flexible.

4.4.1. Bit-Packing in Image File Data Structures

We refer to storing different values in a single datum as bit-packing and to the result as a bit field. This is one of the simplest forms of compression. Using a boolean value as an example, a single byte can store up to seven of them, one boolean state per bit, while boolean data types generally occupy at least 1 byte. Without further trickery, this is particularly useful and popular for so-called feature flags, which are boolean values that indicate support for a feature or a request for specific behaviour. See Section 4.4.2 for sophisticated utilization of this paradigm.

There are two important operation classes for extracting values from bit fields. The first is the binary masking operator `AND` (&). Masking refers to the character of the operands, as the operation applies bit-wise. The other is the binary shift operators left-shift ($\ll$) and right-shift ($\gg$). Rather than utilizing masks, their second operator is a 'shift exponent'. The operations are identical to multiplying or dividing by 2^x respectively for their defined domain (C prohibits shift exponents greater than or equal to the integer type's width). Compared to masks that affect specific bits at specific positions, the shift exponent affects the first operand's value as a whole.

To provide a minimum amount of backward compatibility, all unused bits must be declared as must-be-zero. This way, future specifications can use these bits to invoke new behaviour for non-zero values, without changing the semantics of old binaries.

4.4.2. Utilizing Constraints for Compressed Encoding

In terms of image file formats, as we have seen, there are various constraints on metadata. For example, the data alignment requirement for image segments must be a power of two. The most efficient encoding for this information is

$$\mathsf{encode}(x) = \log_2(x).$$

By storing only the exponent, we can compress a 32-bit image segment data alignment requirement to just its 5-bit exponent. Decoding requires only a single binary operation:

$$\mathsf{decode}(y) = 2^y = 1 \ll y.$$

Another example is that image segments addresses and sizes must be aligned on this data alignment requirement boundary. There is no general requirement for its minimum. However, the PE format specifies a minimum of 512 bytes, and some operating systems reject image files with an alignment less than the platform memory page size [6, 8, 10]. The following assumes an alignment of 4 KiB.

The image segment addresses and sizes can be encoded as they are:

$$\mathsf{encode}(x) = x,$$

but we know that

$$x \bmod 4096 = x \ \& \ 00000\mathrm{FFF}_{16} = 0,$$

which effectively frees the lower 12 bits. When designating bits [12 . . 31] in a 32-bit word to the same bits of the address or size, a single binary operation is sufficient to decode either value:

$$\mathsf{decode}(y) = y - y \bmod 4096 = y \ \& \ \mathrm{FFFFF000}_{16}.$$

The bits [0 . . 11] can now be used for other purposes, such as feature flags. Semantically, this means that the bits [12 . . 31] store the other factor to 4096. The PE format uses this encoding for Base Relocation Blocks (see Section 2.1.3), which store their size in the low 12 bits [6].

4.4.3. Optimized Bit Field Layout

For fixed-width integers, it is obvious to see that the worst-case number of masking bits is exactly the width of the type of the first operand. Meanwhile, the shift exponent only requires a logarithmic number of bits compared to the first operand. Thus, in general, shift operations are easier to encode in machine instructions and can result in shorter machine code (see Table 4.1). Of course, this still depends on the exact circumstances, including the target ISA.

Decoding	**Code Size** [byte]		
	x86-64	**ARM64**	**RISC-V**
Mask-Only	29	40	28
Shift-Only	25	40	24

Table 4.1.: *Bit Field Decoding Code Size Comparison.*

Comparison of code sizes for mask-only and shift-only bit field decoding of two values across architectures. For the source data, see Appendix A.1.

Continuing the previous example of 4 KiB alignment, some data structures can store both the address of a memory page and something related to indexing within it, such as an offset, a size, or a count of elements. For the memory page address, since the (minimum) size for a given target architecture is usually known at build-time, one can factorize the value and store only the other factor, as seen before. When processing these data separately, it is more efficient to store the memory page address low and the other datum high, even if it is an offset and the opposite arrangement seems more natural (seeAppendix A.1).

Flags are a special kind of bit field value because they only store boolean information. All that matters is whether the bit at the queried index is 0 or 1, which some architectures support checking in hardware [11]. Thus, they can be located anywhere in a bit field and always be evaluated with only a single `AND` instruction. However, especially when checking multiple flag bits that are located high at the same time, it may be beneficial to right-shift the bit field first and then test the bits with `AND` operations that now use shorter masks (see Appendix A.1.3). Some ISAs, as a result, allow for shorter instruction encodings for the `AND` operations (see Table 4.2).

In the absence of special optimizations, such as those above, it is generally best to arrange the bit field members in ascending order of size. This naturally reduces the values of the shift exponents. A convenient side effect is that the most significant member, which will be the largest value, does not require a masking operation for decoding.

Decoding	**Code Size** [byte]		
	x86-64	**ARM64**	**RISC-V**
Naive	38	52	32
Shift-Assisted	37	44	32

Table 4.2.: *Bit Field Decoding Code Size Comparison.*

Comparison of code sizes for naive and shift-assisted bit field decoding of three flags. For the source data, see Appendix A.1.3.

4.4.4. Relocation Fixup Table and Chains

For the image relocation, we chose a delta encoding similar to dyld fixup chains (see Section 2.1.2). Replacing traditional image relocation fixups with their chained counterparts makes two assumptions:

1. The image does not need to run as-is, since storing metadata is inherently incompatible with having a valid target value.

2. The image does not need to be relocated again, as applying a chained image relocation fixup destroys its metadata, including the full index of the chained image relocation fixups.

RT drivers violate Assumption 2 because after the operating system determines the virtual memory address mapping of the UEFI runtime memory, they may need to be relocated. This can be solved by indexing the image relocation fixup table during the first load-time image relocation and using this index when relocating again for runtime execution. Due to time constraints, we simply disabled chained image relocation fixups for RT drivers and copy the image relocation table from the image file.

Pre-memory PEIMs violate both Assumptions 1 and 2. Firstly, they must execute from the firmware storage before the main memory is initialized. But they also need to be relocated in case they are shadowed (this counts as a second image relocation, because the first time happened as part of the preloading at build-time). Due to Assumption 1, we cannot store any metadata at the relocation target address and as such, so we need to disable chained image relocation fixups for pre-memory PEIMs.

Similarly to chained image relocation fixups, the head fixup entries can be stored using delta encoding as well. For this, a fixup root entry stores the target address offset to the end of the previous image relocation fixup. For the first fixup root entry, the previous reference address is always 0. It is followed by a flexible array of head fixup entries that store the type of the current and the target offset of the next head reloc entry (see Figure 4.2). In comparison to PE, which uses 4-KiB-divided blocks, this technique can aggregate arbitrarily many head fixups under a single fixup root entry, so long as their offset is close enough. Like with chained image relocation fixups, our implementation allows offsets less than 4095 bytes. Effectively, the image relocation table becomes a nested singly-linked list (see Figure 4.3).

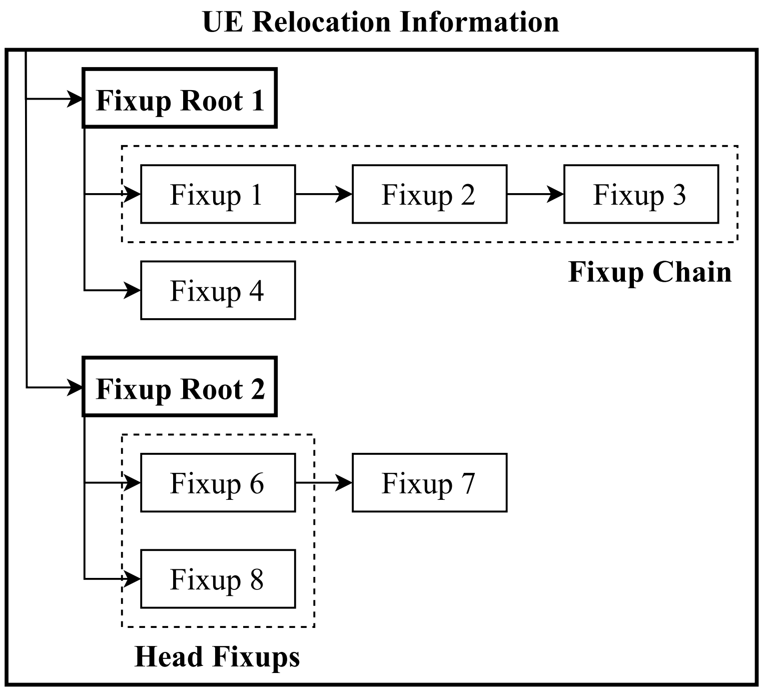

Figure 4.2.: *UE Relocation Information Logical Hierarchy.*

The logical hierarchy of the UE relocation information. Fixup roots point to an area with one or more image relocation fixups. Head fixups are indexed within a fixup root and point to image relocation targets. These targets can, depending on the type, be the start of an image relocation fixup chain, which is a singly-linked list of image relocation targets that technically resides outside the image relocation table.

Furthermore, the image relocation target addresses — this includes both chained image relocation fixups and non-chained head fixups — are ordered ascending. As offset accumulation happens relative to the end of the previous image relocation fixup for either kind (see Figure 4.4), this especially means that they are disjoint and that this can easily be verified in $O(n)$ while parsing the table. This could greatly benefit formalization or verification efforts, as the overall structure becomes much simpler and stricter. Verifying the correctness of deemed infeasible in the given timeframe when we verified a new implementation of the EDK II PE loader [40].

4.4.5. Separating XIP Metadata from the Image Address Space

UEFI XIP executable files reside on the firmware storage. They are a special class of executable files that must be executable without further modification, including image file loading and image relocation. Techniques such as chained image relocation fixup cannot be used, as they rely on storing metadata within the image address space.

In particular, XIP means that image segments must be stored aligned, so the necessary padding must be stored explicitly. If adhering to the platform

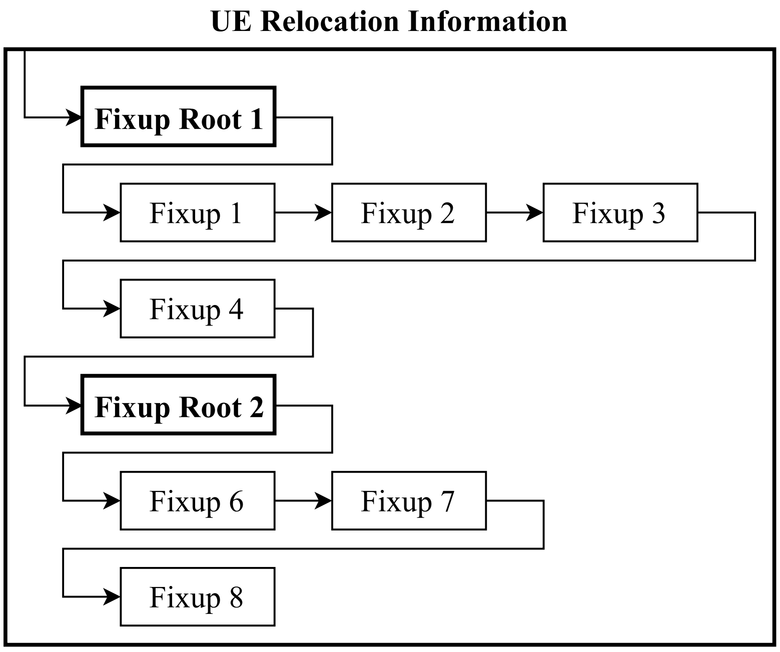

Figure 4.3.: *UE Relocation Information Technical Organization.*

The technical organization of the UE relocation information. All data structures use delta encoding, i.e. they carry offsets to the end of the previous structure. Unlike the logical hierarchy (see Figure 4.2), the technical organization is flat. This encoding is not only more space-efficient, but also forces all fixup roots, head fixups, and chained fixups to be disjoint and appear in ascending order by their target.

memory page size, this can be a significant overhead and can result in several kibibytes of padding. As firmware storage is usually limited, the amount of aligned data should be kept to a minimum.

For the PE format, the image file header, image relocation directory, and similar metadata are part of the image address space. As such, they must be aligned by the image alignment constraint. Since these metadata are not technically part of the image and should always be read-only, they can be stored separately while preserving the general image semantics.

So we propose separating the metadata from the image address space completely. Compared to a non-XIP image, a XIP image file should output two files:

- The metadata file stores the image file header, the image relocation table, the image debug table, and other types of metadata. It only needs to be aligned regarding the general type data alignment constraints and should be protected with read-only memory permissions.

- The image address space file stores the actual image content. The image segments may have interdependent references and as such they cannot be split. Therefore, to respect their potentially different memory permissions,

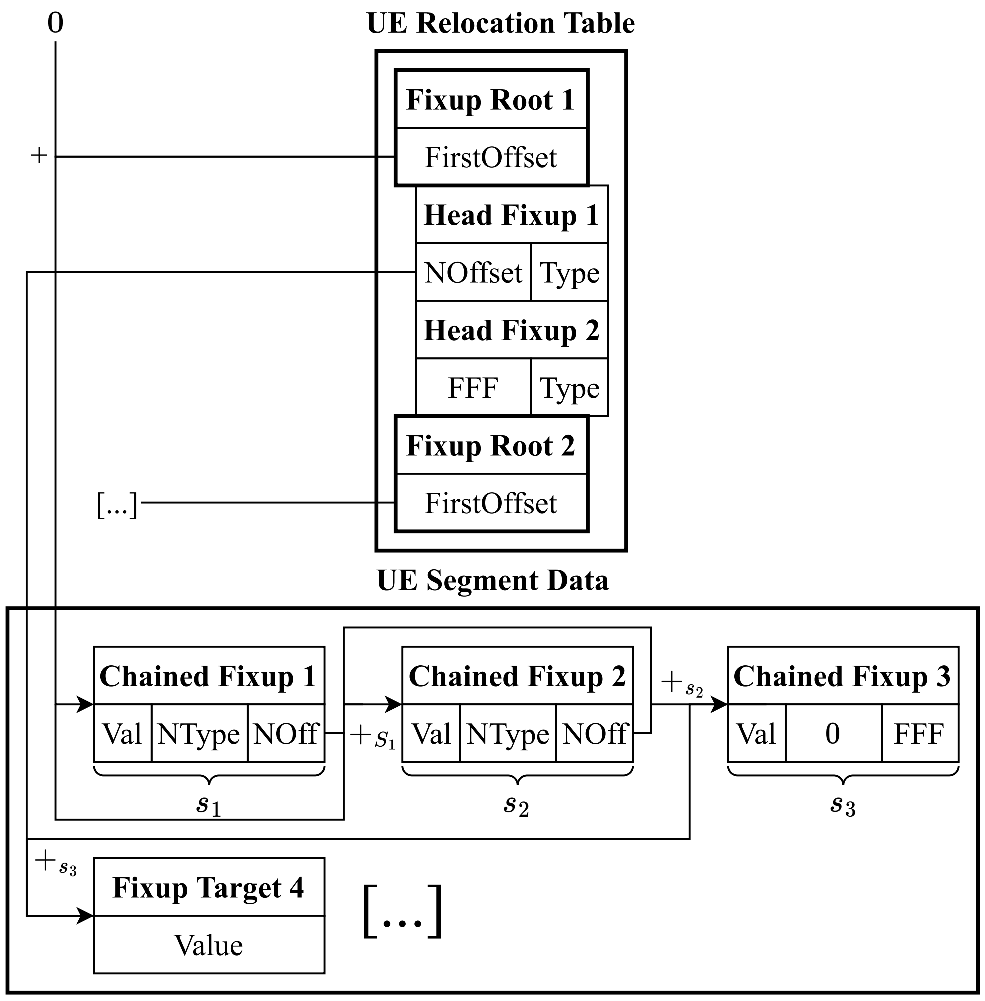

Figure 4.4.: *UE Relocation Information Delta Encoding.*

A more detailed illustration of the technical organization of the UE relocation information (see Figure 4.3). Fixup roots carry the offset of their first head fixup. The offset of the first fixup root is relative to 0 *(so effectively absolute), while for the rest, it is relative to the end of the last image relocation fixup in the previous fixup root. Head fixups store their type and the offset to the next head fixup from the end of the last image relocation fixup in the current chain. Chained fixups store the type and relative offset of the next fixup in the chain, if any. An offset of* FFF_{16} *indicates that there are no more items in the fixup chain or the head fixup list. The offset of the next item accumulates continuously from the offset of the end of the previous item.*

they must be stored aligned on at least the platform memory page size boundary, as if the image were loaded there.

The current firmware file system (FFS) design organizes executable files in a flat manner, which prohibits us from implementing the above scheme (see Section 8.1).

4.5. Encoded Constraints, Simplified Parsing, and Loader Security

Another important aspect of designing an image file format is to keep the data structures easy to parse. In fact, the safest parsing code is the one that does not exist. In this section, we will discuss how smart encoding can reduce the runtime parsing overhead and how to reduce the metadata that are exposed as part of an image address space.

4.5.1. Enforcing Constraints via Compressed Encoding

If well-designed, the use of constraints for compressed encoding has a dual nature. Not only do the constraints enable the compression (see Section 4.4.2), but the compression can also enforce the constraints.

Going back to the original example, the image data alignment is ideally a multiple of the platform memory page size, as otherwise memory permissions cannot be applied (see Section 3.1.2). While different platforms may have varying memory page sizes, UEFI effectively defines a minimum of 4 KiB [3]. Rather than testing whether the value is a multiple of 4 KiB at runtime, we can enforce a constraint using the encoding scheme alone by storing

$$\mathsf{encode}(x) = \log_2(\frac{x}{4096}) = \log_2(x) - 12.$$

This leaves us with a decoding operation of:

$$\mathsf{decode}(y) = 2^y \cdot 4096 = 1 \ll (y + 12).$$

One more example is the memory permissions configuration. By good security practice, we want to enforce WˆX (see Section 3.1.3). With a typical read-write-execute bit field, this yet again requires a runtime check [49]. However, let us take a look at all possible memory permissions configurations:

1. none
2. read
3. write
4. execute
5. read, write
6. read, execute
7. write, execute
8. read, write, execute

Using the W^X rule, we can immediately eliminate Items 7 and 8. Also, Items 1 and 3 are not useful in the context of image files. This leaves us with:

1. read
2. execute
3. read, write
4. read, execute

As apparent, the amount of valid configurations is a power of two. This allows us to encode them using exactly 2 bits by enumerating them. The resulting encoding prevents any violations of the W^X rule from being expressed, without involving runtime checks.

4.5.2. Reimplementing HII Package List Exposure

Due to the parsing burden on the module dispatcher and the overcomplicated structure of the PE resource directory, we propose a novel replacement for the current UEFI HII package list storage. It must meet the following requirements:

1. Due to the technical requirements of Independent BIOS Vendors (IBVs), tools must be able to replace the UEFI HII package list post-build [119].
2. Due to the implementation details of the EDK II Shell, the UEFI HII package list must be parsable at boot-time.
3. To provide a significant security benefit over the current solution, the production code paths must not rely on parsing the UEFI HII package list container.
4. To provide a significant portability benefit, it should not rely on concepts that are not well established among the common executable file formats, ELF, Mach-O, and PE.

The layout of the data within image segments depends on the image file linker that composes it. However, given only a single datum, the common MSVC, LLVM lld, and GNU ld image file linkers will all produce an image segment with that datum located at the very beginning. This behaviour can be exploited by aggregating all UEFI HII package list data into a single C struct, instantiating a global variable from it, and assigning it to a dedicated image file section (which will be the only image file section of a dedicated image segment).

To meet the second requirement and to allow substitution of the UEFI HII package list, which means its size cannot be known at build-time, we store it along with the data (see Source Code 4.3). This way, the size can be updated if the substituting UEFI HII package list is larger or smaller than the one it replaces.

As we are guaranteed this data structure instance resides at the beginning of the designated image segment, the UEFI HII package list can be located, parsed, and replaced with one that is at most the same size. To be able to replace it with a larger one, we force the designated image segment to be at

```
typedef struct {
  UINT32                      Size;
  EFI_HII_PACKAGE_LIST_HEADER Header;
  //
  // UEFI HII package list data.
  //
} MODULE_HII_PACKAGE_LIST;
```

Source Code 4.3.: *Data Structure for the UEFI HII Package List.*

The data structure that encodes the UEFI HII package list in a novel approach. Unlike the conventional UEFI solution, which uses the PE-specific resource directory that requires extra parsing, we utilize a regular C struct instance. To preserve the flexibility of the former method, the data structure is self-contained and forced into a dedicated image segment similarly to before.

the end of the image address space. This allows us to grow and shrink it as required without affecting the rest of the image.

This new model satisfies all the requirements we have been able to gather, without requiring any parsing from the module dispatcher.

4.5.3. Image Debugging Information and Image Address Space

The PE format has a unique property compared to the other image file formats. The image file header is loaded at the image base address and the first image segment starts after the image file header. Other image file formats always load the first image segment at the image base address, and if the image file header is to be loaded, it must be part of the former. Since we generally do not want to load the image file header into the image address space at all, it must be removed during the conversion.

Consequently, when converting an image file to the new executable file format, the image file is first rebased to the image base address minus the offset of the first image segment and then the image base address is reset to the previous value. As a result, the image file header has been removed from the image address space without changing the image base address.

For debugging, this means that the image symbol file will not match the newly converted file. For ELF and Mach-O source files, the source file itself is the image symbol file and as such, they could also have the reserved area for the image file header removed. However, for PE files, debug information is usually stored separately in a program database (PDB) file [120], which is not trivial to rebase due to its proprietary nature [121]. As a workaround that works for all source image file formats, we store the stripped size along with the file path of the image symbol file (see Figure 4.5). When mapping the latter with a debugger, the image base address is adjusted by said stripped size to account for the missing image file header, so that the offsets of the image segment in the image address space match again.

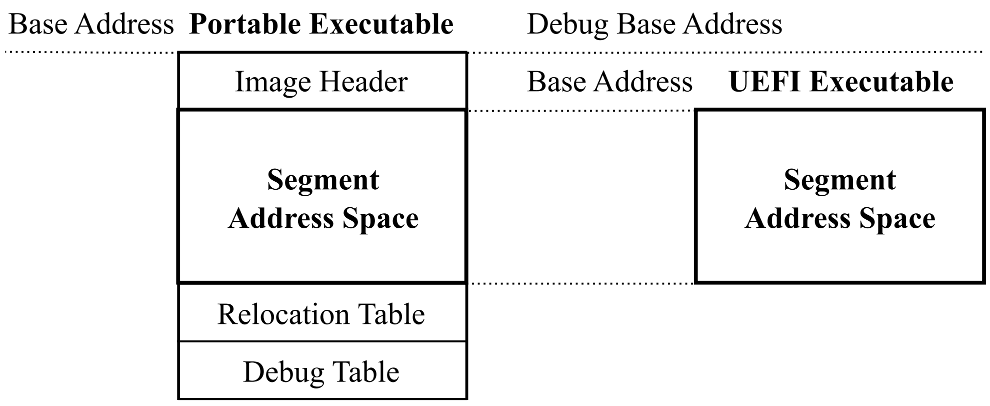

Figure 4.5.: *UE Debug Model.*

The UE debug model. Compared to the source PE image, UE does not map the image file header and the metadata tables. While the latter does not cause any problems, as debugging is limited to symbolication and there can be no image symbols targeting these tables, removing the image file header will move all image segments addresses. This can be accommodated with ELF or Mach-O source files by relocating them, as they carry the image symbols, but PE uses the proprietary Microsoft format PDB for debugging. Since there is no easy way to update it with the new image segment addresses, UE can store a delta to the image base address in the debug table. When symbolicating, this offset is applied and the PDB is loaded so that the image segments align.

4.5.4. Discarding Load-Time Information

It is common for image files to load their image file header and debugging information into the image address space — for the PE format, the former is even done implicitly [6]. This makes it easy to debug an image by having all required data easily discoverable by memory traversal. However, this also has important downsides:

1. This information occupies significant space, especially as all image segments must be aligned at a memory page boundary.
2. This information can be used by an operating system runtime attack on RT drivers to modify them with great reliability and predictability.

Even for production builds, which have most of the useful debugging information stripped, the PE format requires loading the image relocation information into the image address space [6]. While it can be discarded safely after loading and linking have finished in theory, most notably the EDK II PE image file loader does not do this [21].

In response, we have introduced a function at the API level to discard information that is only needed at load-time [49]. For PE files, this overwrites the image relocation and image debug tables with zero-values. Also, as a configuration option to the PE loader, the image file header area can be omitted. Finally, for image debugging, we removed the assumption that the image symbols file path is part of the image address space. Instead, it is retrieved at load-time and stored as part of the central UEFI debug image info table [3].

5. Implementing UE Conversion and Loading

In this chapter, we will discuss the details of the implementation, which consists of a tool for image file format conversion, and the image file loader library. First, we will discuss the choice of an unstable design in Section 5.1. In Section 5.2, we will explain how we used the image abstraction to design a generic UEFI image API. Next, we will give a brief and high-level overview of the image file loading process in Section 5.3. In Section 5.4, we again use the image abstraction to derive a new architecture for converting between image file formats. Finally, we describe how our image file conversion stack validates its outputs in Section 5.5.

5.1. Unstable Design and Scope Limitation

The proposed novel UEFI executable file format has a deliberately unstable design that is subject to change. This allows adjustments to be made during early adoption to accommodate previously unknown use cases without the technical debt resulting from backward compatibility. For many reasons, it may be desirable to preserve the unstable nature of the executable file format for firmware-internal use. With the introduction of forward-edge CFI mitigation support for the PE format, none of the intended ways of extensibility, such as the concept of data directories, was supported by EDK II in a backwards-compatible manner. In response, the revised PE format designates a PE debug entry to expose the necessary metadata to support forward-edge CFI mitigations [6]. This results in unexpected semantics as the previous design designated these entries only for debugging, which typically requires only a single entry [21].

However, for firmware-internal images built from source, there is little reason to support backward compatibility, as they are rebuilt with each new revision of the firmware image. This allows extending the unstable executable file format with new concepts as needed without having to define sophisticated abstractions for extensibility in advance.

5.2. Image File Loading and Image Abstraction

The image abstraction is not only useful for image generation but also image loading. Here, the operations on the image abstraction are most relevant.

5.2.1. Generic UEFI Image File API

Of course, in order not to break the UEFI boot specification and thus backward compatibility, we still need to support PE files. To avoid the burden of supporting UE and PE at the same time on the caller code, and to strengthen the image file loader design, we have created a new, generic library API [122]. Like the design for image generation, it is also based on the UEFI image abstraction we defined (see Section 4.1), and it implements all the generic operations. Many of the metadata can be retrieved using getter functions. We have not abstracted the image segment and image relocation tables. Instead, we provide higher-level abstractions, such as retrieving an ‘image record’, which contains all the information necessary to apply memory permissions to an image address space.

5.2.2. Support for Multiple Image File Formats

Using a generic API for multiple image file formats means that each function needs to distinguish between them for any given reference object. To enable this, we have introduced generic context data structures for load-time and runtime contexts. These start with a format identifier that allows us to efficiently determine the image file format. Our implementation defines another, internal data structure that aggregates pointers to all format-specific library and helper functions, which is statically instantiated for both UE and PE. The generic function implementations use a macro that implements an efficient decision for which of these data structures to use for format-specific calls. For further flexibility, it includes checks for build-time configuration options that define which image file formats are supported. We verified that with all modern toolchains[1], the indirection via the internal data structures is optimized away when link-time optimization (LTO) is enabled. The same is true for any support code for image file formats that have been disabled by said build-time configuration.

5.2.3. Per-Source File Format Support

As declared by the scope of the UE format, it is intentionally unstable and intended for firmware-internal use, so only from the firmware storage. Old UE binaries may remain structurally valid, but a new parser may interpret their semantics differently. This could lead to hidden security vulnerabilities. At the same time, some UEFI PI and UEFI phases will only load image files from the firmware storage, such as PEI and MM. However, with a naive library design, they would still carry the support code for PE files.

For these reasons, we have added configuration options to choose which image file formats are available from either the firmware or external storage. For the scope of UE, this would mean that the firmware storage would only carry UE files, while the external storage would only carry PE files. The caller specifies the source of the image file when initiating the parsing. As with

[1]We tested GCC 11.3, Clang 16.0.6, Xcode 14.3, and MSVC 2022 v17.6.2.

the configuration of supported file formats on the library-internal level (see Section 5.2.2), we have verified that LTO optimizes the unreachable support code away. This solves both problems at once — an attacker cannot introduce old UE binaries to the system to trigger exploitable behaviour, and the module dispatchers that only handle firmware image files do not carry support code for both image file formats. Since the DXE phase deals with both sources, because it provides the image infrastructure for the BDS phase, it must inherently support both image file formats.

5.3. Algorithms for the UE File Format

To illustrate the high-level image file loading process, we provide some pseudo-code descriptions for the main algorithms. While the first two algorithms are specific to the purposed UE file format, the last one is format-independent and part of the generic image file loader library.

Algorithm 1 Loading a UE File.

```
Require: The UE file and a buffer to hold the image address space as input.
Ensure: The input buffer holds the image address space.
1: for every image segment do
2:     Copy the image segment file contents.
3:     Zero the remaining image segment size.
4: end for
5: Zero the remaining data of the input buffer.
```

Algorithm 2 Relocating a UE File.

```
Require: The UE file, the image memory, and the load address as inputs.
Ensure: The image has been relocated to execute from the load address.
1: for every fixup root do
2:     for every head fixup in the root do
3:         for every fixup in the chain do
4:             Apply the image relocation fixup.
5:         end for
6:     end for
7: end for
```

Algorithm 3 Setting up an Executable File for Execution.

```
Require: The executable file as input.
Ensure: The executable file has been loaded and is ready to be executed.
1: Optionally hash (e.g. execute Algorithm 5) and authenticate the executable
   file.
2: Parse the executable file and ensure structural correctness.
3: Allocate enough read-write memory pages to hold the image address space.
4: Load the image (e.g. execute Algorithm 1).
5: Relocate the image (e.g. execute Algorithm 2).
6: for every image segment do
7:     Enforce the requested image segment memory permissions.
8: end for
9: Flush the execution cache for the image address space.
```

5.4. Image Generation and Image Abstraction

Like the current tooling used by EDK II for some platforms, UE files should be generated from other image file formats. To do this, we use the previously defined image abstraction to perform the conversion as such:

1. Convert ELF, Mach-O, PE, and UE image files into the intermediate representation.

2. Perform operations on the intermediate representation (e.g. relocate the image, or convert it into a XIP image).

3. Convert the intermediate representation into an UE file.

The proposed intermediate representation is a manifestation of the image abstraction that we defined to make it easier to convert between specific image file formats (see Figure 5.1 and Section 4.1.1).

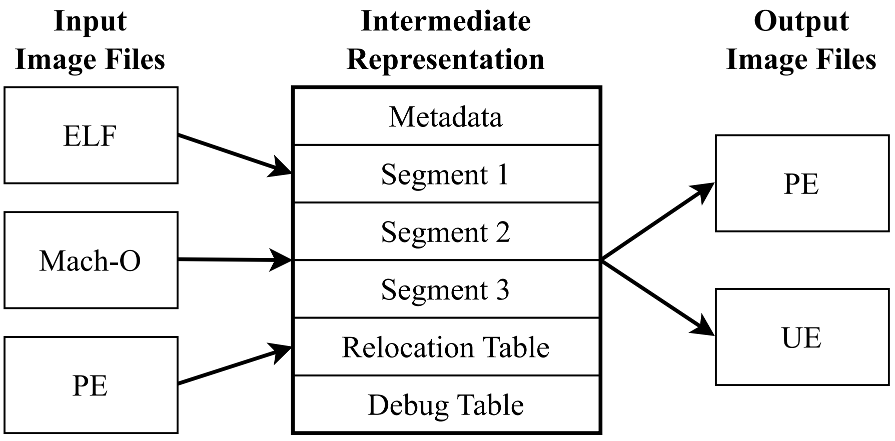

Figure 5.1.: *Image Intermediate Representation.*

The intermediate representation of an image. We derived it from the data structures of the image abstraction we defined earlier (see Section 4.1.1). The high-level idea is to instantiate an intermediate representation from input image file, allow generic operations to be performed on it (e.g., image relocation), and finally to generate an output image file from it.

5.5. Output Validation

To reduce the testing overhead and the risk of exploitable translation bugs, we do as much output validation as possible at build-time as possible. Below we will look at the structural and semantic validation of the translation performed by our image file conversion stack.

5.5.1. Conversion Equivalence Validation

As such, it can be used not only as an intermediate step during the conversion but also as an abstract representation of an image in general. To take advantage of this ability to validate the equivalence of the input image file and the output image file, we propose to make the intermediate representation semi-canonical:

- Most of the metadata is trivially canonical.
- Image segments appear in ascending order by their address.
- Image segment sizes are multiples of the image alignment.
- Image segment names are optional and may be omitted entirely. Even if they are not omitted, they may be truncated by the output conversion. The output name must be a prefix of the input name.
- Any image relocation fixup that is not strictly necessary for image relocation is discarded.
- Image relocation fixups are converted to a universal format. To support all input file formats, only the least common denominator in terms of expressiveness is supported.
- Image relocation fixups appear in ascending order by their target address.

To validate the correctness of the conversation, we can compare the equivalence of the semi-canonical intermediate representations (see Section 5.4) of the input file and the output file (see Figure 5.2). Since the intermediate representation is only semi-canonical, we cannot simply test for equality. In particular, for image segment names, which may be truncated or omitted, we check whether the name in the output file is a prefix of the name in the input file.

5.5.2. Format Constraint Validation

In the past, malformed image files have been shipped by various vendors [42, 41]. The problems range from suboptimal configuration, such as too lax image segment data alignment, to violations of the PE specification, such as misaligned data structures. In order to combat such problems, extensive structural validation of all generated image files must be performed. This should be done using the appropriate boot-time parsing libraries to ensure that the constraints are in sync with the consumer code.

For performance reasons, typical parsing libraries do not verify all requirements, including our proposed UE library and the revised PE library [40]. This may be because the constraint in question does not lead to unsafe behaviour if it is violated, e.g. the order of image relocation fixups. To account for this, we propose to support an optional flag to enable extensive structural validation. This must be disabled in production firmware so as not to degrade performance, but debug firmware may use it at the discretion of the platform maintainer.

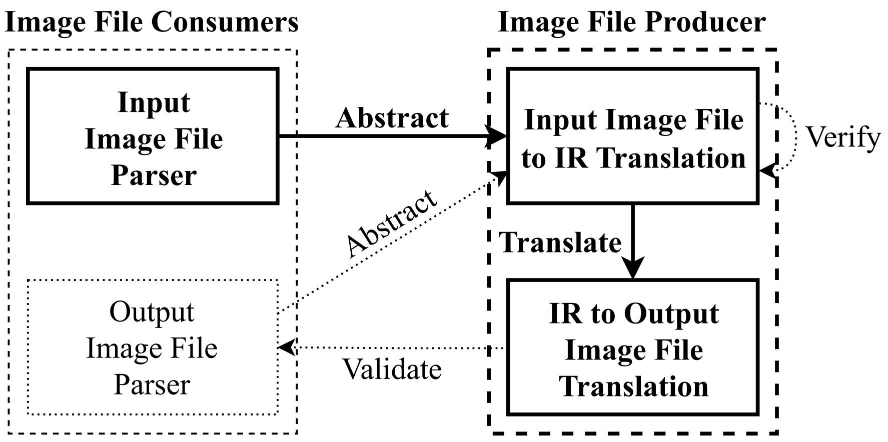

Figure 5.2.: *Image File Conversion Architecture.*

The architecture of our image file conversion stack. First, the input image file parser is used to create the semi-canonical intermediate representation. Operations can now be performed on this intermediate representation. It is then translated into the output image file. This completes the conversion. To verify its correctness, the output file is parsed and abstracted into an intermediate representation. Using the semi-canonical properties, we then compare the input and output representations for equivalence.

Most importantly, after generating an image file at build-time, it must be parsed with the extensive structural validation enabled, to ensure that it is well-formed and meets all the requirements of the consumer.

Reusing the parsing library used at boot-time ensures that the producer code and the consumer code remain in sync. Note that if changes are made to the image parsing code, the entire firmware image must be rebuilt to ensure that all image files pass the extensive structural validation performed by the image file format conversion tool.

6. Results and Discussion

Finally, we will evaluate our work. Sections 6.1 to 6.3 will cover our evaluation of the implementation's safety, functionality, and space efficiency, respectively. To conclude our work, we will outline our steps to enable artifact evaluation in Section 6.4.

6.1. Safety Validation

To ensure the safety of our implementation, we have applied sophisticated analysis methods and performed extensive testing. While this cannot be guaranteed without the use of formal verification techniques, the goal is to ensure safety, especially memory safety in particular, for all conforming and malformed inputs. Due to time constraints, we focused only on PE-to-UE and UE-to-UE conversions for static analysis and fuzz testing. As such, both ELF input parsing and PE output generation have not been validated. We have also excluded libraries provided by the upstream AUDK project from static analysis.

6.1.1. Static Analysis

For static analysis, we used GitHub CodeQL, Clang Static Analyzer, and Coverity. All three tools found valid bugs in both the upstream image file conversion tool and the support logic for the novel executable format. While we squashed fixes for the latter as development artifacts, we fixed the former in the 45440d1 ∼ 4f49f08 commit range of our AUDK fork [122]. The classes of valid bugs discovered were resource leaks and non-terminated strings (not technically a bug, but a welcome precaution). We have never addressed nor inspected reported problems in underlying libraries from AUDK. At the time of writing, no valid bugs in our implementation remain from the reports.

6.1.2. Fuzz Testing

We performed fuzz testing using LLVM libFuzzer [66] and the existing OpenCore infrastructure for userland fuzz testing [54]. Two separate tools cover the parsing library and the conversion tool for the new executable file format, respectively [122]. In addition to the executable file input, the OpenCore fuzz testing infrastructure randomizes the global state for fault injection and platform configuration. To accomplish this, we reserve a configuration region at the end of the fuzz testing input.

OpenCore fuzz testing utilities share the trailer of the input data among the fuzz target input and the configuration region, as this allows to easily add legitimate, real-world inputs to the fuzz testing corpus. Testing the PE parsing library like this yielded good results, as the shared region corresponds to PE debug data that does not affect the control flow much. For the new executable file format, however, data is much more tightly-packed and there is little free-form data. To guarantee good results, we segmented the input data to strictly separate the fuzz target input from the global state configuration. To still use seed the fuzz testing corpus with real-world inputs, we developed a script that appends a global state region to them.

The initial state disables any fault injection, hence all of them should reach an overarching success path. The fuzzer can now mutate the global state configuration, the fault injection in particular, to trigger alternative and error paths, without altering the executable file portion of the input.

Fuzz testing discovered several safety violations, memory leaks, incorrect assertions, and unreachable code. These include the problems fixed by the commit range 45afac4 $\sim$ 69b4dd1 of our AUDK fork [122]. It achieved excellent code coverage metrics in the process (see Table 6.1). We used 8 workers per fuzz testing target for about two weeks and stopped the process when corpus mutation slowed down dramatically.

File	**Coverage** [%]		
	Lines	**Functions**	**Branches**
UE Parsing	100	100	100
UE Conversion	98.1	100	97.3
Total	98.8	100	98.1

Table 6.1.: *UE Implementation Fuzz-Testing Coverage.*

Code coverage for the fuzz testing tools. For the source data, see Appendix D.

6.1.3. Fault Injection into Dynamic Memory Allocation

To trigger all error handling paths, the OpenCore fuzz testing infrastructure performs fault injection on the EDK II memory allocation library functions. At the time of writing, this is done via a bitmap of width 64. While bitmaps allow complex patterns to recur, the executable file format conversion tool aborts execution when dynamic memory allocation fails. As such, failure patterns will not improve the path coverage of the fuzz target. At the same time, the width of the bitmap limits which allocations can be failed. As every allocation failure terminates the tool, at most 64 applications can be failed by having exactly one bit cleared at a time. Due to the use of dynamically-growing buffers, more dynamic allocations may occur, however, they are unable to fail.

To address this issue, we propose a new modulo-based fault injection mode [122]. Instead of failing a dynamic memory allocation when the bit that corresponds to its index is clear, it fails if its index mod N is equal to $N - 1$. This way, the fuzzer can increment the modulus to successively fail every individual dynamic allocation. This exponentially increases the maximum amount of dynamic memory allocations that can be failed compared to the bitmap approach. On the downside, it does not allow for fault injection patterns, but always fails every N-th dynamic memory allocation. As every failure is fatal in our case, this does not cause any issues, though.

To be able to inject faults into every dynamic memory allocation, the growing size of the dynamically-growing buffers was limited to 4 bytes, forcing frequent reallocations. Otherwise, it would be possible that certain operations could never realistically fail, as previous expansion operations would always allocate sufficient memory, but this is infeasible to prove and maintain.

6.1.4. Fuzz Testing Feature Collisions and Fabrication

While monitoring the progress of fuzz testing, we noticed that some error branches for dynamic memory allocation were never hit, despite our best efforts to make them easy to fault one by one (see Appendix D) [122]. In an attempt to seed the corpus with such failure inputs, we manually fabricated a very large number of files with an incrementing fault modulus, which did indeed increase the coverage. However, when we tried to minimize the corpus, the new corpus again had no inputs that triggered these branches.

Although this is not easy to prove, we suspect that these paths were subject to feature collisions, i.e. they were classified as similar or identical to other inputs that did not trigger these branches [123]. The accuracy of fuzz testing code coverage is still an open research problem [124]. However, although they were not added to the corpus, libFuzzer probably generated and successfully processed inputs that triggered these branches.

In order to still trigger most of them with a reasonably sized corpus, we fabricated modules more efficiently [122]. We determined the module with the maximum number of dynamic memory allocations, hoping that it would hit most of the allocation paths, and fabricated exactly as many inputs that fault each allocation. As a result, the coverage of the corpus increased significantly but still did not cover all branches (see Appendix D). However, as the fuzzer likely generated and processed inputs that hit them and the coverage data is otherwise excellent, we did not try to optimize the corpus further.

6.2. Functional Validation

Of course, while safety and security are crucial to low-level software implementations, reliability and correctness are equally important. To validate the functionality of our implementation, we performed various kinds of manual and automated tests. In the following, we will discuss the functionality tests we performed regarding unit and system testing, as well as automated test case generation using fuzz testing.

6.2.1. Unit Testing with Build Artifacts

To test the functionality of the conversion logic, we convert all image files from OVMF and ArmVirtQemu across all supported architectures and targets from PE to UE. The correctness check of the output is part of the conversion procedure itself (see Section 5.5.1) and does not require pre-computed outputs. This implicitly tests the correctness of most parts of the parsing library, as the correctness validation uses it for most parsing tasks (see Section 5.5.2). Rather than providing a separate test infrastructure for the remaining parts of the parsing library, future work should focus on deduplicating the unused parts of the parsing library as a first step (see Section 8.2).

6.2.2. System Testing with Virtual Machines

To test the functionality of the fully integrated system, we adapted the existing boot test infrastructure of AUDK. We adjusted the OVMF and ArmVirtQemu platforms to use our novel executable file format for the DXE (excl. EDK II's *DxeCore*), RT, and MM stages. A script then performs automated tests with resulting firmware images using the following environments [125]:

- **TestConsole:** A simple UEFI application that prints 'Hello World!' to the standard output console [125].
- **TestLinux:** A minimal Linux 6.3 environment that performs basic platform initialization and then prints 'Hello World!' to the standard output console [125].
- **TestWindows:** A Windows 10 Preinstallation Environment that boots to a command prompt [126].

Throughout development, we have had continuous automated testing of these scenarios using GitHub Actions continuous integration and continuous delivery (CI/CD).

6.2.3. Automated Test Case Generation with Fuzz Testing

The image file format conversion performs an internal check on the correctness of the translation output (see Section 5.5.1). Thus, fuzz testing doubles as automated test case generation. The test tool will either reject an input if the input parser rejects it, or it will generate a correct translation. If the equivalence check fails, the tool asserts, which would classify the input as a crash during fuzz testing. However, due to the coverage-guided nature of libFuzzer, it is possible that most of the generated inputs were invalid and therefore rejected much earlier than to realistically stress-test the conversion equivalence logic.

6.3. Space Efficiency Evaluation

To compare the space efficiency of the newly-proposed UE format with the established PE solution, we generated both types of executable files from the OVMF and ArmVirtQemu platforms across architectures and toolchains. Unlike unit tests (see Section 6.2.1), *OvmfIa32X64* was not considered because *OvmfIa32* and *OvmfX64* provide a superset of its image files. We also only compared the `RELEASE` target, as it is the only one relevant to the production stage. Similarly, the *CLANGPDB* toolchain was not considered, because it does not support compilation of Arm image files and could therefore distort the final results between the architectures.

While the results are not groundbreaking, this is to be expected, as we focused mainly on optimizing the image file header and the image relocation table, which are generally small compared to the size of the image segments, especially without dynamic linking (see Tables 6.2 to 6.4). However, considering SPI firmware remains sparse and real-world platforms can be much more modular than OVMF and ArmVirtQemu, total platform space-savings of up to 78 KiB are respectable. Once XIP is implemented for UE (see Section 4.4.5), we expect the results to improve significantly when comparing sizes on the final firmware image.

Platform	**Arch**	**Toolchain**	**Saving** [%]		
			Min.	**Avg.**	**Max.**
OVMF	IA32	GCC5	0.17	10.50	50.00
		CLANGDWARF	0.17	10.66	53.75
	X64	GCC5	0.58	9.53	47.92
		CLANGDWARF	0.60	10.09	47.83
ArmVirtQemu	ARM	GCC5	0.17	9.60	49.43
		CLANGDWARF	0.17	8.70	50.60
	AARCH64	GCC5	0.60	9.52	54.41
		CLANGDWARF	0.41	9.01	56.25
Average			0.36	9.70	51.27

Table 6.2.: *Platform PE-to-UE Relative Space-Saving.*

Relagtive module space-savings from the PE-to-UE conversion per platform, architecture, and toolchain. All modules have been compiled in `RELEASE` *mode. For the source data, see Appendix C.*

6.4. Artifact Evaluation and Reproducibility

To ensure the reproducibility of our results, we released a Docker [127] container image [128] with the development and testing environment used to produce all artifacts and to perform all tests [122]. Furthermore, we set up a GitHub Actions CI/CD [129] infrastructure to automate artifact generation and all tests based on the existing AUDK infrastructure. We used GitHub Actions CI/CD to produce all discussed results, which did not involve self-hosted runners or

Platform	**Arch**	**Toolchain**	**Saving** [byte]		
			Min.	**Avg.**	**Max.**
OVMF	IA32	GCC5	560	672.24	1,376
		CLANGDWARF	536	669.80	1,384
	X64	GCC5	504	744.61	4,920
		CLANGDWARF	432	758.93	4,960
ArmVirtQemu	ARM	GCC5	528	649.32	1,208
		CLANGDWARF	552	666.64	1,376
	AARCH64	GCC5	472	753.00	4,952
		CLANGDWARF	480	768.36	4,760
Average			508.0	710.36	3,117.0

Table 6.3.: *Platform PE-to-UE Absolute Space-Saving.*

Absolute module space-savings from the PE-to-UE conversion per platform, architecture, and toolchain. All modules have been compiled in `RELEASE` *mode. For the source data, see Appendix C.*

Platform	**Arch**	**Toolchain**	**Saving**	
			[KiB]	[%]
OVMF	IA32	GCC5	66.96	2.64
		CLANGDWARF	66.72	2.60
	X64	GCC5	76.35	2.63
		CLANGDWARF	77.82	2.78
ArmVirtQemu	ARM	GCC5	53.90	2.79
		CLANGDWARF	55.34	2.31
	AARCH64	GCC5	64.71	2.56
		CLANGDWARF	66.03	1.87
Average			65.98	2.49

Table 6.4.: *Platform PE-to-UE Total Space-Saving.*

Total space-savings from the PE-to-UE conversion per platform, architecture, and toolchain. All modules have been compiled in `RELEASE` *mode. For the source data, see Appendix C.*

proprietary code.

To increase confidence in the results, we also released a container description file that should continue to produce Docker images close to the image we provided. Unfortunately, bit-by-bit reproducibility is currently not a core feature of Docker and solutions like repro-get [130] proved to be suboptimal for Ubuntu-based container base images. As a best-effort approach, we provide a Docker container image description that pinned all versions of all packages installed.

7. Conclusion

In order to significantly reduce the complexity of runtime parsing of UEFI executable files, we have proposed a novel format that is strictly tailored to UEFI firmware requirements. We have shown that files of our new format provide considerable space-saving when used for all platform modules, and their internal organization is visibly simpler compared to PE. In particular, it significantly extends the benefits of TE while retaining the expressiveness of PE in a UEFI context.

In addition, to make image file conversion easier to maintain, extend, and validate, we have implemented an approach based on an intermediate representation. This allows input and output formats to be processed independently of each other. At the same time, operations such as relocating an image became format-independent, contributing to improved extensibility. Because we designed the intermediate representation to be semi-canonical, we were able to use it to reduce the validation of the output file to parsing the input file and comparing the resulting intermediate representation to the original one. With a small caveat, we did not encounter any malformed output files (see Section 8.2), as the conversion tool correctly aborted if the conversion was incorrect.

Finally, we applied static analysis and fuzz testing to our implementation. Due to the equivalence checking performed by the conversion procedure, the latter covered both safety and functional testing. After fixing the initial set of bugs, neither approach uncovered any new problems, and at the same time, we encountered no problems during real-world testing.

We subjected all results to artifact evaluation [122]. This includes pre- and post-conversion executable files, AUDK firmware images for virtual machines, static analysis reports, and the fuzz testing corpora along with their code coverage. To further ensure reproducibility, GitHub Actions CI/CD performed all tests using a semi-reproducible container. We have included both the container image used and the corresponding build description in the supplementary material.

All in all, we have solved the problems outlined in the problem statement and have empirically demonstrated the achievements in space efficiency and security hardening (see Section 1.6).

8. Future Work

We would now like to discuss further improvements to our implementation that could be made in the future, as well as suggest ways to extend the core ideas to a larger scale.

8.1. Missing Features and Architectures

For various reasons, even though we have solved the problem statement, we have not been able to provide all the features and compatibility one might expect. Below we will go through the shortcomings of our implementation and the reasons why we have not been able to address them.

- **XIP:** At this time, the planned XIP mechanisms (see Section 4.4.5) cannot be implemented, because the current EDK II build system generates all PE files as XIP and later stages rely on this fact. This is ensured by GenFw (see Section 2.5.1). For PE files that do not meet memory page size alignment, the difference in space requirement is negligible, and the parser does not need to distinguish between XIP and non-XIP. As for the novel executable file format, we proposed a strict distinction, there is no good way to integrate XIP support at this time. This implies that neither SEC nor PEI are currently supported.

- **DxeCore:** EDK II's *DxeCore* currently tries to re-parse its own image file header to gather the information for image address space memory protection. This works for PE files because their image file header is always loaded implicitly into the image address space. For the novel executable file format, we proposed to not do so (see Section 4.5.4). The issue can easily be fixed by having *PeiCore* pass the necessary information to *DxeCore* via Hand-Off Blocks (HOBs). However, we did not implement this due to a novel patch to introduce memory protection into the PEI phase [131]. Its introduction would implicitly resolve the issue.

- **RISC-V, LoongArch, forward-edge CFI:** As we targeted the EDK II fork AUDK, we did not implement support for the RISC-V and LoongArch ISAs, or for forward-edge CFI, as AUDK does not support any of them at the time of writing. Supporting the two ISAs only requires porting their ISA-specific image relocation fixup to the novel executable file format, while forward-edge CFI only requires translating a support flag, assuming PE support for all of them is added first.

8.2. Deduplication of Image Relocation Table Parsing

Image relocation fixups parsing can vary widely between image file formats, even if the high-level concepts are very similar. For this reason, it is not easy to provide an efficient abstraction for it. Unfortunately, this led us to duplicate the logic into the conversion tool, to not regress the performance and size of the parsing library.

However, similarly to multi-format and multi-source support (see Sections 5.2.2 and 5.2.3), we believe it might be possible to develop an abstraction API that is well-optimized by the compiler. If that was feasible, the code duplication could be removed and the risk of deviating logic is removed. This would further strengthen the result of output equivalence checking (see Section 5.5.1). Unfortunately, due to time constraints, we were unable to develop such an abstraction that satisfied our optimization criteria.

8.3. Module Inter-Linking

UEFI continues to embrace a very modular architecture, factoring different functionalities into separate drivers, much like operating systems kernels do (terminology may vary, e.g. for macOS they are called kernel extensions). However, UEFI and EDK II in particular even separate security-critical components like the CPU abstraction used to apply memory permissions and generic platform policies like the supported image authentication technologies [2, 21]. The former makes EDK II downstreams prone to load order-related bugs, the latter requires preserving API stability for no particular reason (and thus makes it hard to reuse an existing context structure that represents an image file, as the structure is subject to change). The obvious solution is to attempt to merge modules where beneficial.

8.3.1. Static Linking, Pre-Ordering, and Events Listeners

Static linking is the simplest technique for merging modules. One approach is to design modules as static libraries and manually invoke their initialization routines as needed. Another approach is to utilize EDK II library constructors as entry points and let the EDK II build system automatically order the constructors according to the static library dependency lists.

The main disadvantage of using static linking is the reliance on static libraries (which are often just a collection of object files). They are generally not cross-compatible with other compilers and sometimes not even between compiler versions. This introduces a dependency on the module vendor's build setup, whereas a separate driver is always interoperable.

Another issue that may require changes to the dependency's code is depex. Since PPIs and UEFI protocols can be installed at will, cross-dependencies are possible, and there is no general way to tell where a particular load order will satisfy each module's dependencies. Arguably, cross-dependencies are a bad design choice and it is not typical to use depex to depend on PPIs and UEFI protocols that are only installed on demand. By avoiding both and annotating each producer with the PPIs or UEFI protocols it installs in its entry point, it is possible to compute a correct load order at build-time.

If the code cannot be changed, the behaviour of the module dispatcher can be mimicked using event listeners. With a PPI or UEFI protocol installation notification. Once all dependencies are available, the entry point can be invoked.

8.3.2. Prelinking and Composite Modules

To avoid the drawbacks of static linking, an interesting approach to dynamic linking can be considered. In the past, Apple shipped a full-fledged dynamic linker in its XNU kernel, called KXLD [8]. As dynamic linking can be a complicated process and both performance and security are valid concerns, they developed pre-linked kernel images. Kernel Collections are the latest iteration. The main instance is essentially a composite module of a kernel and many kernel extensions needed to boot (see Figure 8.1). The secondary instances, on the other hand, still depend on the main instance via pre-linking.

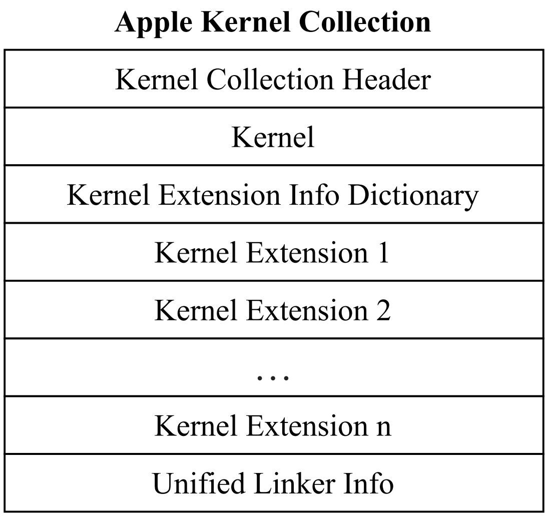

Figure 8.1.: *Kernel Collection Executable File Format.*

A simplified overview of the Apple Kernel Collection format. It aggregates an operating system kernel and any number of kernel extensions into one, unified kernel collection executable file. The dynamic linker creates a wrapper Mach-O image that has the components shown above as image segments. The kernel extension info dictionaries (they contain information like which devices to attempt attaching to) are aggregated into one, unified kernel extension info dictionary (also known as 'prelink info'). The kernel and all kernel extensions of the Kernel Collection image have their dynamic linker info (e.g. image relocation fixups) aggregated into one, unified info image segment for the dynamic linker. This process does not require access to the source code or object files of the involved modules.

The composition process involves merging both the image segment tables and the image relocation tables of all components. As this fixes the offsets between the kernel's image segments and those of all its drivers, dynamic linking can be moved to the build-time by resolving the image symbols and issuing image relocation fixups in return. The result essentially mimics static linking, except for features like image segment merging and LTO. Apple solved the former for the dyld shared cache, which consists of user-space applications and shared libraries, by issuing image relocation fixups for relative addressing.

In order for the kernel to be able to locate and start its drivers, the Kernel Collection contains a file with their locations and other metadata in a special image file section. Unlike static linking with pre-ordering, locating and launching the dependency entry points is a runtime operation. However, this could be addressed by supporting the generation of an entry point stub that replaces the original entry point.

8.4. Image File and Asset Digital Signing

Since UE is currently only intended for firmware-internal use, which generally does not require per-file authentication, we have not designed a working solution for it. However, if the new format proves to be useful, it may make sense to support it for external modules, such as OS loaders. For this event, there are three possible ways to handle authentication.

8.4.1. Appended Digital Signatures and File Prefix Hashing

One way is to embed the digital signature in the image files. For example, it can be appended to it (see Algorithm 4) [55, 51]. This is similar to how Authenticode works, but collapses the hashing algorithm into a single step. This is done by storing the size of the unsigned image file in the image file header, which is available at build-time. This is also the offset of the digital signature, if any, and its size is the size of the entire image file minus the stored offset. This does not require any modification to the image file, the digital signature is simply appended.

Algorithm 4 Hashing an Image File With an Appended Digital Signature.

```
Require: The image file, a cryptographic hash function as input.
Ensure: The image cryptographic hash buffer has the image file digital signature.
1: Cryptographic hash the range $[0..\texttt{UnsignedSize} - 1]$ of the image file.
```

8.4.2. External Digital Signatures

Another approach to storing digital signatures is externally from the file they sign. This is employed by Apple both for their firmware-level and kernel-space (i.e. Apple Image4), as well user-space (i.e. Apple application bundles) designs. The main advantages are that the signed file remains unchanged and that the filesystem takes over some parsing burden from the image file loader. Depending on the cryptographic hashing algorithm, the signed file does not need to be parsed at all to authenticate it (see Algorithm 5).

Algorithm 5 Hashing a UE File With an External Signature.

```
Require: The executable file, a cryptographic hash function as input.
Ensure: The image cryptographic hash buffer has the UE file digital signature.
 1: Apply the cryptographic hash function to the entire executable file.
```

8.4.3. Container Digital Signatures

Similar to the embedded digital signatures, a container format could be specified where the digital signature is essentially prepended. This has the advantage that the signed data does not need to be parsed in any way prior to authentication. It can also be implemented as a generic format that is independent of the data being signed. Most importantly, it could be used for any asset that an image file might require and, in theory the authentication of the file could be part of loading the file from secondary storage. However, such standardization is rather unrealistic and using external digital signatures to collectively sign all assets in a database is likely to be more efficient and flexible.

8.5. Self-Relocation

EDK II for Arm architectures requires another concept that we have not yet covered — self-relocation [21]. This is used for scenarios like TrustZone, where the hardware maps a module to a load address that is unknown at build-time. If this module has image relocation fixups, it must process them itself, as the hardware only maps the image file. It is critical that any code that runs until the image relocation has succeeded is not subject to image relocation and does not use any affected data.

There are currently no precautions in place to ensure a safe self-relocation process [21]. One way to help establish them is to identify all the code and data required for self-relocation and move them to dedicated image file sections. With the critical components isolated, the image relocation fixups can be checked at build-time to ensure that they do not target any of them. Furthermore, when preserving the object file relocation fixups, which may contain sources of relative addressing, it can be checked that no critical component addresses a non-critical component. If both conditions are met, the self-relocation process is safe.

8.6. Implementing Safety-Critical Code in Rust

Despite the well-known caveats about using Rust for embedded development (see Section 2.7.2), there are also organizational factors. First, EDK II is a much more corporate product than the Linux kernel. As a lot of development happens in proprietary environments that may not have a Rust development environment, and it may require formal requests to provide one. Also, a look at the corresponding mailing lists shows that the Linux kernel has much more development traffic than EDK II. The slow progress of proposals to add Rust support to EDK II is a clear indicator of both a lack of serious interest and possibly a lack of audience. Firmware developers often have an electrical engineering background, which still resolves around C, and many may not be familiar with Rust or its novel concepts. For these reasons, we have chosen not to involve Rust in our implementation, in order to provide the easiest way to integrate our work into production firmware.

Still, the strong guarantees of the Rust compiler would be a valuable addition to our stack of safeguards. While it is likely to be a long time before Rust properly enters the firmware domain, we consider it to be a worthwhile investment on the road to memory safety.

A. Encoding Space Efficiency Analyses

This appendix contains sample source code and corresponding machine code. We used the Compiler Explorer[1] to generate the machine code.

A.1. Bit Field Encoding Formats

In this section, we will present machine code generated for different types of bit field decoding. In general, shift-based bit field decoding generates smaller code compared to mask-based bit field decoding. Adjacent flag decoding is well-optimized across compilers and architectures.

A.1.1. Mask-Only Bit Field Decoding

In this section, we will present machine code generated for mask-only bit field decoding.

Source Code A.1.: *Mask-Only Bit Field Decoding.*

```c
int main(void) {
  //
  // Input: An address encoded as follows.
  // [Bits 11:0]  The offset into the memory page.
  // [Bits 31:12] The index of the memory page.
  //
  volatile unsigned int addr;
  //
  // Output: The base address of the memory page.
  //
  volatile unsigned int paddr = addr & 0xFFFFF000U;
  //
  // Output: The offset into the memory page.
  //
  volatile unsigned int poff = addr & 0x00000FFFU;
  return 0;
}
```

[1]https://godbolt.org (visited on 14/07/2023)

Source Code A.2.: *Machine Code for Source Code A.1 (x86-64).*

Compiler:	*x64 msvc v19.33*
Compiler Options:	*/O2*
Code Size:	*29 bytes*

```
0140001000    mov    eax, DWORD PTR [rsp+0x8]
0140001004    and    eax, 0xfffff000
0140001009    mov    DWORD PTR [rsp+0x8], eax
014000100d    mov    eax, DWORD PTR [rsp+0x8]
0140001011    and    eax, 0xfff
0140001016    mov    DWORD PTR [rsp+0x8], eax
014000101a    xor    eax, eax
014000101c    ret
014000101d
```

Source Code A.3.: *Machine Code for Source Code A.1 (ARM64).*

Compiler:	*ARM64 gcc 12.2*
Compiler Options:	*-O3*
Code Size:	*40 bytes*

```
0000400540    sub    sp, sp, #0x10
0000400544    mov    w0, #0x0
0000400548    ldr    w1, [sp, #4]
000040054c    and    w1, w1, #0xfffff000
0000400550    str    w1, [sp, #8]
0000400554    ldr    w1, [sp, #4]
0000400558    and    w1, w1, #0xfff
000040055c    str    w1, [sp, #12]
0000400560    add    sp, sp, #0x10
0000400564    ret
0000400568
```

Source Code A.4.: *Machine Code for Source Code A.1 (RISC-V).*

Compiler:	*RISC-V rv64gc clang 15.0.0*
Compiler Options:	*-O3*
Code Size:	*28 bytes*

```
0000000684    addi    sp, sp, -16
0000000686    lwu     a0, 12(sp)
000000068a    lui     a1, 0xfffff
000000068c    and     a0, a0, a1
000000068e    sw      a0, 8(sp)
0000000690    lwu     a0, 12(sp)
0000000694    slli    a0, a0, 0x34
0000000696    srli    a0, a0, 0x34
0000000698    sw      a0, 4(sp)
000000069a    li      a0, 0
000000069c    addi    sp, sp, 16
000000069e    ret
00000006a0
```

A.1.2. Shift-Only Bit Field Decoding

In this section, we will present machine code generated for shift-only bit field decoding.

Source Code A.5.: *Shift-Only Bit Field Decoding.*

```
int main(void) {
  //
  // Input: An address encoded as follows.
  // [Bits 19:0]  The index of the memory page.
  // [Bits 31:20] The offset into the memory page.
  //
  volatile unsigned int addr;
  //
  // Output: The base address of the memory page.
  //
  volatile unsigned int paddr = addr << 12U;
  //
  // Output: The offset into the memory page.
  //
  volatile unsigned int poff = addr >> 20U;
  return 0;
}
```

Source Code A.6.: *Machine Code for Source Code A.5 (x86-64).*

Compiler: *x64 msvc v19.33*
Compiler Options: */O2*
Code Size: *25 bytes*

```
0140001000    mov    eax, DWORD PTR [rsp+0x8]
0140001004    shl    eax, 0xc
0140001007    mov    DWORD PTR [rsp+0x8], eax
014000100b    mov    eax, DWORD PTR [rsp+0x8]
014000100f    shr    eax, 0x14
0140001012    mov    DWORD PTR [rsp+0x8], eax
0140001016    xor    eax, eax
0140001018    ret
0140001019
```

Source Code A.7.: *Machine Code for Source Code A.5 (ARM64).*

Compiler:	*ARM64 gcc 12.2*
Compiler Options:	*-O3*
Code Size:	*40 bytes*

```
0000400540    sub    sp, sp, #0x10
0000400544    mov    w0, #0x0
0000400548    ldr    w1, [sp, #4]
000040054c    lsl    w1, w1, #12
0000400550    str    w1, [sp, #8]
0000400554    ldr    w1, [sp, #4]
0000400558    lsr    w1, w1, #20
000040055c    str    w1, [sp, #12]
0000400560    add    sp, sp, #0x10
0000400564    ret
0000400568
```

Source Code A.8.: *Machine Code for Source Code A.5 (RISC-V).*

Compiler:	*RISC-V rv64gc clang 15.0.0*
Compiler Options:	*-O3*
Code Size:	*24 bytes*

```
0000000684    addi   sp, sp, -16
0000000686    lw     a0, 12(sp)
0000000688    slliw  a0, a0, 0xc
000000068c    sw     a0, 8(sp)
000000068e    lwu    a0, 12(sp)
0000000692    srli   a0, a0, 0x14
0000000694    sw     a0, 4(sp)
0000000696    li     a0, 0
0000000698    addi   sp, sp, 16
000000069a    ret
000000069c
```

A.1.3. Flag Decoding and Compiler Optimizations

In this section, we will present machine code generated for flag decoding.

Source Code A.9.: *Naive Bit Field Flag Decoding.*

```c
int main(void) {
  volatile unsigned int xyz;
  volatile unsigned int x = (xyz & 0x80000000U) != 0;
  volatile unsigned int y = (xyz & 0x40000000U) != 0;
  volatile unsigned int z = (xyz & 0x20000000U) != 0;
  return 0;
}
```

Source Code A.10.: *Machine Code for Source Code A.9 (x86-64).*

Compiler:	*x64 msvc v19.33*
Compiler Options:	*/O2*
Code Size:	*38 bytes*

```
0000401000    push   ecx
0000401001    mov    eax, DWORD PTR [esp]
0000401004    shr    eax, 0x1f
0000401007    mov    DWORD PTR [esp], eax
000040100a    mov    eax, DWORD PTR [esp]
000040100d    shr    eax, 0x1e
0000401010    and    eax, 0x1
0000401013    mov    DWORD PTR [esp], eax
0000401016    mov    eax, DWORD PTR [esp]
0000401019    shr    eax, 0x1d
000040101c    and    eax, 0x1
000040101f    mov    DWORD PTR [esp], eax
0000401022    xor    eax, eax
0000401024    pop    ecx
0000401025    ret
0000401026
```

Source Code A.11.: *Machine Code for Source Code A.9 (ARM64).*

Compiler:	*ARM64 gcc 12.2*
Compiler Options:	*-O3*
Code Size:	*52 bytes*

```
0000400540    sub     sp, sp, #0x10
0000400544    mov     w0, #0x0
0000400548    ldr     w1, [sp]
000040054c    lsr     w1, w1, #31
0000400550    str     w1, [sp, #4]
0000400554    ldr     w1, [sp]
0000400558    ubfx    x1, x1, #30, #1
000040055c    str     w1, [sp, #8]
0000400560    ldr     w1, [sp]
0000400564    ubfx    x1, x1, #29, #1
0000400568    str     w1, [sp, #12]
000040056c    add     sp, sp, #0x10
0000400570    ret
0000400574
```

Source Code A.12.: *Machine Code for Source Code A.9 (RISC-V).*

Compiler:	*RISC-V rv64gc clang 15.0.0*
Compiler Options:	*-O3*
Code Size:	*32 bytes*

```
0000000684    addi    sp, sp, -16
0000000686    lwu     a0, 12(sp)
000000068a    srli    a0, a0, 0x1f
000000068c    sw      a0, 8(sp)
000000068e    lw      a0, 12(sp)
0000000690    slli    a0, a0, 0x21
0000000692    srli    a0, a0, 0x3f
0000000694    sw      a0, 4(sp)
0000000696    lw      a0, 12(sp)
0000000698    slli    a0, a0, 0x22
000000069a    srli    a0, a0, 0x3f
000000069c    sw      a0, 0(sp)
000000069e    li      a0, 0
00000006a0    addi    sp, sp, 16
00000006a2    ret
00000006a4
```

Source Code A.13.: *Shift-Assisted Bit Field Flag Decoding.*

```c
int main(void) {
  volatile unsigned int xyz;
  unsigned int xyz2 = xyz >> 29U;
  volatile unsigned int x = (xyz2 & 0x04U) != 0;
  volatile unsigned int y = (xyz2 & 0x02U) != 0;
  volatile unsigned int z = (xyz2 & 0x01U) != 0;
  return 0;
}
```

Source Code A.14.: *Machine Code for Source Code A.13 (x86-64).*

Compiler: *x64 msvc v19.33*
Compiler Options: */O2*
Code Size: *37 bytes*

```
0140001000    mov    ecx, DWORD PTR [rsp+0x8]
0140001004    shr    ecx, 0x1d
0140001007    mov    eax, ecx
0140001009    shr    eax, 0x2
014000100c    mov    DWORD PTR [rsp+0x8], eax
0140001010    mov    eax, ecx
0140001012    shr    eax, 1
0140001014    and    ecx, 0x1
0140001017    and    eax, 0x1
014000101a    mov    DWORD PTR [rsp+0x8], eax
014000101e    xor    eax, eax
0140001020    mov    DWORD PTR [rsp+0x8], ecx
0140001024    ret
0140001025
```

Source Code A.15.: *Machine Code for Source Code A.13 (ARM64).*

Compiler:	*ARM64 gcc 12.2*
Compiler Options:	*-O3*
Code Size:	*44 bytes*

```
0000400540    sub    sp, sp, #0x10
0000400544    mov    w0, #0x0
0000400548    ldr    w1, [sp]
000040054c    lsr    w2, w1, #31
0000400550    str    w2, [sp, #4]
0000400554    ubfx   x2, x1, #30, #1
0000400558    str    w2, [sp, #8]
000040055c    ubfx   x1, x1, #29, #1
0000400560    str    w1, [sp, #12]
0000400564    add    sp, sp, #0x10
0000400568    ret
000040056c
```

Source Code A.16.: *Machine Code for Source Code A.13 (RISC-V).*

Compiler:	*RISC-V rv64gc clang 15.0.0*
Compiler Options:	*-O3*
Code Size:	*32 bytes*

```
0000000684    addi   sp, sp, -16
0000000686    lwu    a0, 12(sp)
000000068a    srli   a1, a0, 0x1f
000000068e    sw     a1, 8(sp)
0000000690    slli   a1, a0, 0x21
0000000694    srli   a1, a1, 0x3f
0000000696    sw     a1, 4(sp)
0000000698    slli   a0, a0, 0x22
000000069a    srli   a0, a0, 0x3f
000000069c    sw     a0, 0(sp)
000000069e    li     a0, 0
00000006a0    addi   sp, sp, 16
00000006a2    ret
00000006a4
```

A.2. Position Independence and Relative Addressing

In this section, we will compare machine code generated for local non-preemptible image symbol references between PIEs and non-PIEs. Some cases of relative addressing are restricted to PIEs — non-PIEs may resort to absolute addressing for performance reasons.

Source Code A.17.: *Local Non-Preemptible Image Symbol Reference.*

```c
static volatile int var = 0;

static int __attribute__ ((noinline)) func(void) {
    return var;
}

int main(void) {
    return func();
}
```

Source Code A.18.: *Machine Code for Source Code A.17 (x86-64).*

Compiler: *x86-64 gcc 12.2*
Left Compiler Options: *-O3 -fno-pie -mcmodel=large -Wl,-no-pie*
Right Compiler Options: *-O3 -fPIE -Wl,-pie*

```
func:
movabs  eax,ds:0x0
    R_X86_64_64
        .bss
ret
main:
movabs  rax,0x0
    R_X86_64_64
        .text
jmp     rax
```

```
func:
mov      eax,DWORD PTR [rip+0x0]
    R_X86_64_PC32
        .bss-0x4
ret
main:
jmp    5
    R_X86_64_PC32
        .text-0x4
```

Source Code A.19.: *Machine Code for Source Code A.17 (ARM).*

Compiler: *ARM gcc 12.2*
Left Compiler Options: *-O3 -fno-pie -Wl,-no-pie*
Right Compiler Options: *-O3 -fPIE -Wl,-pie*

```
func:
movw    r3, #0                func:
    R_ARM_THM_MOVW_ABS_NC     ldr     r3, [pc, #4]
        .LANCHOR0             add     r3, pc
movt    r3, #0                ldr     r0, [r3, #0]
    R_ARM_THM_MOVT_ABS        bx      lr
        .LANCHOR0             .word   0x00000002
ldr     r0, [r3, #0]              R_ARM_REL32 .bss
bx      lr                    main:
main:                         b.w     0
b.w     0                         R_ARM_THM_JUMP24 func
    R_ARM_THM_JUMP24 func
```

Source Code A.20.: *Machine Code for Source Code A.17 (ARM64).*

Compiler: *ARM64 gcc 12.2*
Compiler Options 1: *-O3 -fno-pie -Wl,-no-pie*
Compiler Options 2: *-O3 -fPIE -Wl,-pie*
Both generated identical code.

```
func:
adrp    x0, 0
    R_AARCH64_ADR_PREL_PG_HI21 .bss
ldr     w0, [x0]
    R_AARCH64_LDST32_ABS_LO12_NC .bss
ret
main:
b       0
    R_AARCH64_JUMP26 .text
```

Source Code A.21.: *Machine Code for Source Code A.17 (RISC-V).*

Compiler: *RISC-V rv64gc gcc 12.2.0*
Left Compiler Options: *-O3 -fno-pie -Wl,-no-pie*
Right Compiler Options: *-O3 -fPIE -Wl,-pie*

```
func:
lui      a5,0x0
    R_RISCV_HI20 var
    R_RISCV_RELAX *ABS*
lw       a0,0(a5)
    R_RISCV_LO12_I var
    R_RISCV_RELAX *ABS*
ret
main:
auipc    t1,0x0
    R_RISCV_CALL func
    R_RISCV_RELAX *ABS*
jr       t1
```

```
func:
auipc    a0,0x0
    R_RISCV_PCREL_HI20
        .LANCHOR0
    R_RISCV_RELAX *ABS*
lw       a0,0(a0)
    R_RISCV_PCREL_LO12_I
        .L0
    R_RISCV_RELAX *ABS*
ret
main:
auipc    t1,0x0
    R_RISCV_CALL func
    R_RISCV_RELAX *ABS*
jr       t1
```

B. UEFI Executable File Format (UE) Specification

This appendix contains a draft for the specification of the UEFI Executable (UE) format [122].

B.1. Basic Data Types

Table B.1.: *Definition of the Basic Data Types.*

Type	**Size** [byte]	**Max. Data Alignment** [byte]
`uint8_t`	1	1
`uint16_t`	2	2
`uint32_t`	4	4
`uint64_t`	8	8

Note: The data alignment requirement follows the natural data alignment model, thus is to be understood as an upper bound, not an exact value.

B.2. UE Header

Table B.2.: *Definition of the UE Machine Identifiers.*

Value	**Description**
0	IA32
1	X64
2	ARM
3	AARCH64
4	RISCV32
5	RISCV64
6	RISCV128

Table B.3.: *Definition of the UE Subsystem Identifiers.*

Value	Description
0	Application
1	Boot Services Driver
2	Runtime Driver

Table B.4.: *Definition of the UE Header (`UE_HEADER`).*

Type	Description
`uint16_t`	The file magic number to identify the UE file format. Must be 'UE'.
`uint8_t`	Information about the image kind and supported architectures.
Bits 2:0	Indicates the subsystem.
Bits 7:3	Indicates the supported architectures.
`uint8_t`	Information about the UE load tables and segments.
Bits 2:0	The number of UE load tables.
Bits 7:3	The index of the last segment in the UE segment table.
`uint32_t`	Indicates the offset of the UE entry point in the UE address space.
`uint64_t`	Information about the UE image.
Bits 51:0	The base UEFI page of the UE image, i.e., the base address in 4 KiB units.
Bits 56:52	Reserved for future use. Must be zero.
Bit 57	Indicates whether the UE image is designated for a fixed address.
Bit 58	Indicates whether the UE relocation table has been stripped.
Bit 59	Indicates whether UE chained fixups are used.
Bits 63:60	The shift exponent, offset by -12, for the UE segment alignment in bytes.
`UE_SEGMENT[]`	The UE segment table. It contains all data of the UE address space.
`UE_LOAD_TABLE[]`	The UE load tables. They contain data useful for UE loading.

B.3. UE Segment Table

All UE segments are contiguous in the UE address space. The offset of the first UE segment in the UE address space is 0. All UE segments' data are contiguous in the UE file. The offset of the first UE segment in the UE file is the end of the UE file header.

Table B.5.: *Definition of the UE Segment Permissions Configurations.*

Value	Description
0	Execute
1	Read, Execute
2	Read, Write
3	Read

Table B.6.: *Definition of a UE Segment Header (`UE_SEGMENT`).*

Type	Description
`uint32_t`	Information about the UE segment in the UE address space.
Bits 19:0	The size, in 4-KiB units, of the UE segment in the UE address space.
Bits 21:20	The UE segment permissions.
Bits 31:22	Reserved for future use. Must be zero.
`uint32_t`	The size, in bytes, of the UE segment in the UE file.

B.4. UE Load Tables

All UE load tables are contiguous in the UE file. The offset of the first UE load table in the UE file is the end of the last UE segment in the UE file. All UE load tables are ordered ascending by their identifier.

Table B.7.: *Definition of the UE Load Table Identifiers.*

Value	Description
0	Relocation Table
1	Debug Table

Table B.8.: *Definition of a UE Load Table Header (`UE_LOAD_TABLE`).*

Type	Description
`uint32_t`	Information about the UE load table.
Bits 28:0	The size, in 8-byte units, of the UE load table in the UE file.
Bits 31:29	The identifier of the UE load table.

B.5. UE Relocation Load Table

Table B.9.: *Definition of the Generic UE Relocation Identifiers.*

Value	Description
0	Absolute 32-bit
1	Absolute 64-bit

Table B.10.: *Definition of the Common Header of UE Chained Relocation Fixups (`UE_RELOC_FIXUP_HDR`).*

Type	Description
`uint16_t`	
Bits 3:0	The relocation type of the next chained relocation fixup. Only valid when [Bits 15:4] are not `0x0FFF`.
Bits 15:4	The offset to the next chained relocation fixup from the end of the current one. If `0x0FFF`, the current chain is terminated. Consult the fixup root for further relocation fixups.

Table B.11.: *Definition of the Generic 64-Bit UE Chained Relocation Fixup (`UE_RELOC_FIXUP_64`).*

Type	Description
`uint64_t`	
Bits 15:0	The common header of UE chained relocation fixups.
Bits 47:16	The address value to relocate.
Bits 63:48	Must be zero.

Table B.12.: *Definition of a UE Fixup Root (`UE_FIXUP_ROOT`).*

Type	Description
`uint32_t`	The offset of the first head fixup, in bytes, from the end of the previous UE relocation fixup (chained or not). The first UE fixup root is relative to 0.
`uint16_t[]` Bits 3:0 Bits 15:4	The head fixups of the UE fixup root. The type of the UE relocation fixup. The offset of the next UE head fixup from the end of the last UE relocation fixup in the chain (if chained). If `0x0FFF`, the current fixup root is terminated.

B.6. UE Debug Load Table

Table B.13.: *Definition of the UE Debug Table Header (`UE_DEBUG_TABLE`).*

Type	Description
`uint8_t` Bits 1:0 Bits 7:2	Information about the image regarding the symbols file. The offset, in image alignment units, to be subtracted from the UE base address in order to retrieve the UE symbols base address. Reserved for future use. Must be zero.
`uint8_t`	The length, in bytes, of the UE symbols path (excluding the terminator).
`uint8_t[]`	The UE symbols path. Must be \0-terminated.
`uint8_t[8][]`	The UE segment name table. Each entry must be \0-terminated. The order matches the UE segment table.

C. UE File Format Space-Saving

This appendix contains the space-saving reports for converting all modules of OVMF and ArmVirtQemu from PE to UE. All modules have been compiled in `RELEASE` mode. It was generated by a custom script and is publicly available and reproducible for artifact evaluation [122].

Table C.1.: *OVMF (IA32, GCC5) PE-to-UE Size Comparison.*

Module	**Size** [KiB]		**Saving** [%]
	PE	**UE**	
StatusCodeHandlerPei	1.25	0.62	50.00
WatchdogTimer	1.44	0.75	47.83
DpcDxe	1.44	0.85	40.76
IncompatiblePciDeviceSupportDxe	1.62	1.01	37.98
MonotonicCounterRuntimeDxe	1.75	1.09	37.50
ReportStatusCodeRouterPei	1.81	1.14	37.07
CapsuleRuntimeDxe	1.75	1.10	37.05
Metronome	1.75	1.11	36.61
NullMemoryTestDxe	1.75	1.14	34.82
StatusCodeHandlerRuntimeDxe	1.81	1.23	31.90
EnglishDxe	2.00	1.40	30.08
CpuIo2Dxe	2.25	1.65	26.74
SecurityStubDxe	2.50	1.87	25.31
EmuVariableFvbRuntimeDxe	2.88	2.24	22.01
BootGraphicsResourceTableDxe	2.50	1.95	21.88
LocalApicTimerDxe	2.62	2.07	21.13
ReportStatusCodeRouterRuntimeDxe	3.12	2.52	19.50
ResetSystemRuntimeDxe	3.19	2.58	19.12
VirtioPciDeviceDxe	3.19	2.59	18.87
SataController	3.69	3.05	17.16
SmbiosPlatformDxe	3.50	2.92	16.52
IoMmuDxe	4.00	3.36	16.02
VirtioRngDxe	3.81	3.21	15.78
SioBusDxe	4.06	3.44	15.38
S3Resume2Pei	4.62	3.95	14.53
FvbServicesRuntimeDxe	4.50	3.88	13.89
QemuKernelLoaderFsDxe	5.12	4.41	13.87
RuntimeDxe	4.44	3.84	13.56
ConPlatformDxe	5.38	4.72	12.21

VirtioBlkDxe	4.75	4.20	11.51
QemuRamfbDxe	6.12	5.46	10.84
PcRtc	6.94	6.22	10.36
VirtioScsiDxe	6.19	5.56	10.10
DiskIoDxe	6.06	5.47	9.79
PcdPeim	6.88	6.23	9.32
Virtio10	7.19	6.55	8.91
ScsiBus	6.44	5.87	8.86
DxeIpl	6.69	6.11	8.64
SmbiosDxe	7.88	7.21	8.43
UsbMassStorageDxe	8.69	8.00	7.91
PciHotPlugInitDxe	7.12	6.56	7.89
ArpDxe	9.69	9.00	7.10
PcdDxe	8.69	8.11	6.65
FaultTolerantWriteDxe	9.38	8.76	6.58
QemuFwCfgAcpiPlatform	9.06	8.49	6.29
Ps2KeyboardDxe	10.75	10.09	6.18
S3SaveStateDxe	10.38	9.73	6.17
VirtioNetDxe	9.50	8.93	6.00
AtaBusDxe	11.12	10.54	5.27
SecMain	12.88	12.20	5.22
PartitionDxe	12.94	12.27	5.19
UdfDxe	13.31	12.64	5.05
DriverHealthManagerDxe	14.19	13.49	4.90
GraphicsConsoleDxe	13.06	12.44	4.78
UhciDxe	12.31	11.73	4.76
PlatformDxe	13.31	12.68	4.75
LogoDxe	12.88	12.27	4.67
VirtioGpuDxe	13.06	12.45	4.67
UsbKbDxe	12.50	11.92	4.62
EbcDxe	13.31	12.71	4.52
AcpiTableDxe	15.44	14.76	4.40
TerminalDxe	15.62	14.95	4.35
VlanConfigDxe	16.62	15.91	4.32
ConSplitterDxe	16.12	15.43	4.31
EhciDxe	14.31	13.70	4.26
SnpDxe	17.75	17.02	4.09
PciSioSerialDxe	15.94	15.29	4.07
UsbBusDxe	14.19	13.62	4.02
QemuVideoDxe	15.62	15.02	3.90
PciHostBridgeDxe	19.62	18.95	3.42
LinuxInitrdDynamicShellCommand	19.06	18.41	3.40
MnpDxe	20.62	19.95	3.26
Fat	20.44	19.78	3.21
AtaAtapiPassThruDxe	23.50	22.76	3.16
NvmExpressDxe	22.44	21.73	3.13

Udp4Dxe	21.19	20.53	3.10
Dhcp4Dxe	23.50	22.78	3.06
PlatformPei	22.06	21.41	2.97
VirtioFsDxe	20.00	19.41	2.97
Mtftp4Dxe	23.31	22.68	2.71
PeiCore	23.19	22.57	2.66
VariableRuntimeDxe	26.56	25.87	2.62
BootScriptExecutorDxe	26.81	26.13	2.53
CpuMpPei	27.69	26.99	2.51
XhciDxe	23.69	23.09	2.51
tftpDynamicCommand	27.75	27.08	2.42
RamDiskDxe	26.06	25.44	2.40
ScsiDisk	27.06	26.41	2.40
DevicePathDxe	31.94	31.27	2.08
TcpDxe	37.50	36.73	2.04
httpDynamicCommand	35.19	34.48	2.00
PciBusDxe	36.75	36.10	1.76
UefiPxeBcDxe	50.06	49.31	1.50
Ip4Dxe	51.88	51.15	1.40
SetupBrowser	58.38	57.59	1.34
DisplayEngine	54.12	53.42	1.30
BdsDxe	62.62	61.84	1.26
HiiDatabase	73.31	72.56	1.02
IScsiDxe	90.25	89.48	0.86
UiApp	98.06	97.30	0.78
DxeCore	117.69	116.81	0.74
Shell	768.88	767.53	0.17
Average	24.90	24.24	10.50
Total	2,539.81	2,472.85	2.64

Table C.2.: *OVMF (IA32, CLANGDWARF) PE-to-UE Size Comparison.*

Module	**Size** [KiB]		**Saving** [%]
	PE	**UE**	
StatusCodeHandlerPei	1.25	0.58	53.75
WatchdogTimer	1.44	0.77	46.74
DpcDxe	1.44	0.83	42.39
CapsuleRuntimeDxe	1.62	0.98	39.42
ReportStatusCodeRouterPei	1.69	1.05	37.96
NullMemoryTestDxe	1.75	1.10	37.05
IncompatiblePciDeviceSupportDxe	1.56	0.98	37.00
MonotonicCounterRuntimeDxe	1.75	1.12	36.16
Metronome	1.50	0.98	34.90
StatusCodeHandlerRuntimeDxe	1.75	1.20	31.70

EnglishDxe	1.94	1.33	31.45
CpuIo2Dxe	2.25	1.62	27.78
SecurityStubDxe	2.50	1.87	25.31
BootGraphicsResourceTableDxe	2.56	1.94	24.39
LocalApicTimerDxe	2.50	1.90	24.06
EmuVariableFvbRuntimeDxe	2.75	2.11	23.30
ResetSystemRuntimeDxe	3.00	2.36	21.35
ReportStatusCodeRouterRuntimeDxe	3.00	2.43	19.01
SmbiosPlatformDxe	3.44	2.79	18.86
VirtioPciDeviceDxe	3.31	2.73	17.45
IoMmuDxe	3.69	3.05	17.16
VirtioRngDxe	3.94	3.28	16.67
SataController	3.69	3.11	15.68
S3Resume2Pei	4.06	3.44	15.38
SioBusDxe	4.06	3.46	14.81
QemuKernelLoaderFsDxe	4.81	4.14	13.96
VirtioBlkDxe	5.00	4.33	13.44
RuntimeDxe	4.31	3.76	12.86
FvbServicesRuntimeDxe	4.44	3.87	12.85
ConPlatformDxe	5.56	4.95	11.10
QemuRamfbDxe	6.12	5.45	10.97
VirtioScsiDxe	6.19	5.55	10.23
PcdPeim	6.62	5.97	9.91
DxeIpl	6.25	5.63	9.88
ScsiBus	6.31	5.71	9.53
DiskIoDxe	6.44	5.83	9.47
PcRtc	6.25	5.67	9.25
Virtio10	7.12	6.48	9.10
PciHotPlugInitDxe	6.31	5.76	8.79
SmbiosDxe	7.81	7.18	8.10
UsbMassStorageDxe	9.12	8.41	7.88
PcdDxe	8.69	8.05	7.28
QemuFwCfgAcpiPlatform	8.94	8.35	6.56
S3SaveStateDxe	10.25	9.58	6.55
ArpDxe	9.88	9.26	6.25
VirtioNetDxe	9.44	8.85	6.21
FaultTolerantWriteDxe	9.44	8.85	6.21
Ps2KeyboardDxe	10.25	9.66	5.79
AtaBusDxe	11.12	10.51	5.55
DriverHealthManagerDxe	14.31	13.58	5.13
PartitionDxe	12.44	11.82	4.96
GraphicsConsoleDxe	13.50	12.84	4.86
VirtioGpuDxe	13.12	12.49	4.82
UhciDxe	12.56	11.96	4.79
AcpiTableDxe	15.19	14.46	4.78
UsbBusDxe	14.44	13.75	4.76

EbcDxe	12.50	11.91	4.69
PlatformDxe	13.44	12.81	4.65
UdfDxe	13.31	12.72	4.46
LogoDxe	12.81	12.24	4.45
UsbKbDxe	12.56	12.01	4.42
PciSioSerialDxe	16.06	15.37	4.33
ConSplitterDxe	16.25	15.56	4.23
EhciDxe	14.69	14.07	4.20
TerminalDxe	15.88	15.22	4.13
QemuVideoDxe	16.56	15.92	3.87
SnpDxe	17.62	16.95	3.86
VlanConfigDxe	16.62	16.00	3.76
PciHostBridgeDxe	18.62	17.95	3.65
LinuxInitrdDynamicShellCommand	17.94	17.31	3.48
PlatformPei	19.62	18.95	3.46
Fat	20.38	19.71	3.26
MnpDxe	21.38	20.71	3.11
AtaAtapiPassThruDxe	23.44	22.72	3.07
NvmExpressDxe	22.75	22.06	3.02
Udp4Dxe	21.06	20.43	3.00
VirtioFsDxe	20.94	20.34	2.87
XhciDxe	24.56	23.88	2.77
Mtftp4Dxe	23.56	22.92	2.72
Dhcp4Dxe	23.31	22.70	2.65
RamDiskDxe	26.62	25.92	2.64
PeiCore	25.75	25.08	2.61
VariableRuntimeDxe	27.19	26.50	2.53
SecMain	25.31	24.68	2.50
tftpDynamicCommand	27.31	26.64	2.46
CpuMpPei	26.62	25.99	2.38
BootScriptExecutorDxe	30.12	29.41	2.36
ScsiDisk	28.50	27.87	2.22
DevicePathDxe	32.56	31.89	2.06
httpDynamicCommand	35.06	34.34	2.05
PciBusDxe	37.00	36.34	1.77
TcpDxe	38.56	37.90	1.72
Ip4Dxe	51.69	50.93	1.47
UefiPxeBcDxe	50.12	49.48	1.29
SetupBrowser	60.56	59.78	1.29
DisplayEngine	57.88	57.16	1.23
BdsDxe	63.31	62.56	1.18
HiiDatabase	76.69	75.86	1.08
IScsiDxe	91.06	90.27	0.87
UiApp	99.81	98.99	0.82
DxeCore	108.81	108.00	0.75
Shell	772.50	771.15	0.17

Average	25.13	24.48	10.66
Total	2,563.62	2,496.91	2.60

Table C.3.: *OVMF (X64, GCC5) PE-to-UE Size Comparison.*

Module	**Size** [KiB]		**Saving** [%]
	PE	**UE**	
StatusCodeHandlerPei	1.12	0.59	47.92
WatchdogTimer	1.44	0.79	45.11
DpcDxe	1.56	0.93	40.50
NullMemoryTestDxe	1.69	1.02	39.35
MonotonicCounterRuntimeDxe	1.62	0.99	38.94
ReportStatusCodeRouterPei	1.88	1.24	33.75
IncompatiblePciDeviceSupportDxe	1.81	1.22	32.76
EnglishDxe	2.25	1.54	31.60
CapsuleRuntimeDxe	1.69	1.16	31.02
StatusCodeHandlerRuntimeDxe	1.75	1.23	29.46
Metronome	2.25	1.61	28.47
SecurityStubDxe	2.88	2.20	23.37
BootGraphicsResourceTableDxe	2.88	2.24	22.01
EmuVariableFvbRuntimeDxe	2.81	2.25	20.00
VirtioPciDeviceDxe	3.75	3.09	17.71
VirtioRngDxe	4.31	3.57	17.21
CpuIo2Dxe	4.25	3.54	16.73
LocalApicTimerDxe	3.75	3.12	16.67
ReportStatusCodeRouterRuntimeDxe	3.75	3.13	16.46
SioBusDxe	4.69	3.96	15.50
SataController	4.19	3.55	15.11
ResetSystemRuntimeDxe	4.31	3.68	14.67
RuntimeDxe	4.44	3.79	14.61
VirtioBlkDxe	5.25	4.57	12.95
SmbiosPlatformDxe	4.94	4.34	12.18
QemuKernelLoaderFsDxe	6.31	5.56	11.88
AmdSevDxe	5.56	4.95	11.10
ConPlatformDxe	6.25	5.59	10.62
S3Resume2Pei	6.69	6.01	10.16
VirtioScsiDxe	6.81	6.13	9.98
DiskIoDxe	6.69	6.05	9.58
TdxDxe	6.50	5.88	9.50
PcdPeim	7.88	7.16	9.13
ScsiBus	6.94	6.30	9.12
DxeIpl	7.38	6.72	8.90
Virtio10	8.25	7.53	8.71
FvbServicesRuntimeDxe	7.19	6.57	8.59

QemuRamfbDxe	7.88	7.20	8.53
PcRtc	7.94	7.31	7.87
SmbiosDxe	9.38	8.73	6.92
UsbMassStorageDxe	10.50	9.78	6.85
PcdDxe	9.81	9.16	6.69
IoMmuDxe	9.75	9.12	6.41
VirtioNetDxe	10.62	9.95	6.40
S3SaveStateDxe	12.12	11.37	6.25
EbcDxe	12.81	12.03	6.10
ArpDxe	11.38	10.69	6.04
PciHotPlugInitDxe	8.56	8.07	5.75
Ps2KeyboardDxe	12.56	11.88	5.41
FaultTolerantWriteDxe	10.19	9.64	5.37
UdfDxe	13.50	12.78	5.32
AtaBusDxe	13.00	12.32	5.23
GraphicsConsoleDxe	14.50	13.78	4.96
VirtioGpuDxe	14.19	13.48	4.96
QemuFwCfgAcpiPlatform	10.62	10.10	4.93
LogoDxe	12.94	12.30	4.89
PartitionDxe	13.19	12.55	4.86
UsbKbDxe	13.94	13.28	4.71
DriverHealthManagerDxe	15.12	14.42	4.65
UhciDxe	14.50	13.84	4.58
ConSplitterDxe	18.81	17.98	4.40
EhciDxe	17.00	16.26	4.37
UsbBusDxe	16.88	16.15	4.31
PciSioSerialDxe	17.06	16.34	4.26
PlatformDxe	13.69	13.11	4.22
VlanConfigDxe	18.06	17.33	4.07
TerminalDxe	17.56	16.85	4.05
QemuVideoDxe	17.88	17.17	3.93
AcpiTableDxe	16.88	16.24	3.75
LinuxInitrdDynamicShellCommand	19.81	19.07	3.75
SnpDxe	18.12	17.46	3.66
PciHostBridgeDxe	17.81	17.23	3.25
PeiCore	24.69	23.89	3.23
Udp4Dxe	23.50	22.79	3.03
AtaAtapiPassThruDxe	26.19	25.40	3.01
MnpDxe	23.94	23.23	2.97
DevicePathDxe	36.69	35.61	2.94
Mtftp4Dxe	25.94	25.18	2.92
Fat	23.25	22.58	2.89
NvmExpressDxe	23.94	23.25	2.87
CpuMpPei (MpInitLibUp)	27.88	27.09	2.83
VirtioFsDxe	21.44	20.85	2.73
tftpDynamicCommand	29.12	28.35	2.66

Dhcp4Dxe	27.88	27.15	2.61
XhciDxe	26.69	26.02	2.52
PlatformPei	28.94	28.21	2.51
RamDiskDxe	27.81	27.14	2.42
ScsiDisk	29.69	29.00	2.32
VariableRuntimeDxe	30.62	29.92	2.30
httpDynamicCommand	37.00	36.23	2.07
SecMain	37.56	36.80	2.02
BootScriptExecutorDxe	39.75	38.98	1.95
CpuMpPei	42.94	42.11	1.93
PciBusDxe	39.69	38.99	1.75
TcpDxe	43.19	42.46	1.68
UefiPxeBcDxe	56.31	55.58	1.30
DisplayEngine	56.88	56.16	1.26
Ip4Dxe	60.94	60.22	1.18
SetupBrowser	64.81	64.12	1.07
BdsDxe	71.69	70.94	1.05
HiiDatabase	83.75	83.02	0.87
DxeCore	132.81	131.68	0.85
IScsiDxe	99.88	99.09	0.79
UiApp	109.38	108.63	0.68
Shell	826.81	822.01	0.58
Average	27.60	26.87	9.53
Total	2,897.81	2,821.46	2.63

Table C.4.: *OVMF (X64, CLANGDWARF) PE-to-UE Size Comparison.*

Module	**Size** [KiB]		**Saving** [%]
	PE	**UE**	
WatchdogTimer	1.44	0.75	47.83
StatusCodeHandlerPei	1.06	0.55	47.79
MonotonicCounterRuntimeDxe	1.56	0.93	40.50
DpcDxe	1.50	0.92	38.54
NullMemoryTestDxe	1.56	0.98	37.00
CapsuleRuntimeDxe	1.56	1.02	34.50
EnglishDxe	2.06	1.35	34.47
ReportStatusCodeRouterPei	1.75	1.16	33.48
IncompatiblePciDeviceSupportDxe	1.69	1.13	32.87
StatusCodeHandlerRuntimeDxe	1.62	1.15	29.33
Metronome	2.12	1.54	27.57
BootGraphicsResourceTableDxe	2.75	2.05	25.57
EmuVariableFvbRuntimeDxe	2.69	2.05	23.55
SecurityStubDxe	2.62	2.02	23.21
VirtioPciDeviceDxe	3.69	2.86	22.46

LocalApicTimerDxe	3.69	2.95	20.13
ResetSystemRuntimeDxe	4.06	3.34	17.88
CpuIo2Dxe	4.00	3.31	17.19
SataController	3.94	3.28	16.67
ReportStatusCodeRouterRuntimeDxe	3.62	3.05	15.73
VirtioRngDxe	4.00	3.39	15.23
SioBusDxe	4.31	3.66	15.22
RuntimeDxe	4.25	3.62	14.71
VirtioBlkDxe	4.94	4.27	13.61
SmbiosPlatformDxe	4.56	3.95	13.36
QemuKernelLoaderFsDxe	6.00	5.20	13.28
S3Resume2Pei	6.19	5.40	12.75
ConPlatformDxe	5.94	5.20	12.50
AmdSevDxe	5.00	4.38	12.50
TdxDxe	6.31	5.60	11.26
VirtioScsiDxe	6.25	5.56	11.00
PcdPeim	7.38	6.57	10.91
DiskIoDxe	6.62	5.95	10.26
QemuRamfbDxe	7.44	6.68	10.19
DxeIpl	6.88	6.18	10.11
Virtio10	7.75	6.97	10.08
FvbServicesRuntimeDxe	7.00	6.35	9.26
ScsiBus	6.75	6.12	9.26
PcdDxe	9.19	8.40	8.59
PcRtc	7.19	6.59	8.37
SmbiosDxe	9.06	8.35	7.84
UsbMassStorageDxe	9.81	9.05	7.72
EbcDxe	12.12	11.26	7.15
VirtioNetDxe	9.94	9.24	7.00
IoMmuDxe	9.50	8.84	6.99
ArpDxe	10.62	9.92	6.62
Ps2KeyboardDxe	11.81	11.09	6.15
PartitionDxe	11.75	11.03	6.12
AtaBusDxe	12.06	11.34	5.96
S3SaveStateDxe	11.31	10.64	5.94
QemuFwCfgAcpiPlatform	10.25	9.66	5.79
PciHotPlugInitDxe	7.81	7.39	5.40
FaultTolerantWriteDxe	9.56	9.05	5.39
LogoDxe	12.94	12.25	5.31
VirtioGpuDxe	13.69	12.98	5.14
DriverHealthManagerDxe	14.94	14.17	5.13
UhciDxe	13.88	13.17	5.07
UdfDxe	12.69	12.05	4.99
UsbBusDxe	15.56	14.80	4.87
UsbKbDxe	13.00	12.37	4.87
GraphicsConsoleDxe	14.44	13.75	4.76

AcpiTableDxe	15.94	15.19	4.71
TerminalDxe	16.81	16.02	4.69
ConSplitterDxe	17.38	16.57	4.63
PciSioSerialDxe	16.50	15.80	4.26
PlatformDxe	13.88	13.30	4.17
QemuVideoDxe	18.19	17.44	4.12
EhciDxe	16.19	15.54	4.01
SnpDxe	18.00	17.30	3.91
VlanConfigDxe	17.38	16.70	3.87
PeiCore	26.81	25.83	3.67
LinuxInitrdDynamicShellCommand	18.19	17.53	3.61
PciHostBridgeDxe	17.44	16.82	3.54
DevicePathDxe	34.44	33.23	3.49
AtaAtapiPassThruDxe	24.62	23.80	3.36
Fat	22.06	21.34	3.26
NvmExpressDxe	23.00	22.28	3.12
Mtftp4Dxe	24.75	23.99	3.06
Dhcp4Dxe	26.19	25.41	2.98
Udp4Dxe	22.69	22.02	2.96
CpuMpPei (MpInitLibUp)	26.62	25.84	2.93
MnpDxe	23.38	22.71	2.84
VirtioFsDxe	20.88	20.33	2.62
XhciDxe	25.75	25.09	2.58
RamDiskDxe	27.00	26.30	2.58
PlatformPei	27.94	27.23	2.52
tftpDynamicCommand	27.62	26.96	2.40
VariableRuntimeDxe	29.56	28.88	2.30
httpDynamicCommand	35.94	35.18	2.11
ScsiDisk	29.56	28.95	2.06
BootScriptExecutorDxe	42.44	41.57	2.04
SecMain	42.06	41.20	2.04
CpuMpPei	41.31	40.48	2.02
PciBusDxe	37.19	36.49	1.87
TcpDxe	41.00	40.26	1.81
UefiPxeBcDxe	53.12	52.38	1.40
Ip4Dxe	56.75	55.97	1.38
SetupBrowser	63.62	62.82	1.26
DisplayEngine	58.00	57.27	1.25
BdsDxe	68.25	67.52	1.06
HiiDatabase	82.19	81.41	0.95
DxeCore	121.44	120.42	0.84
IScsiDxe	94.06	93.35	0.76
UiApp	104.00	103.25	0.72
Shell	806.62	801.78	0.60
Average	26.63	25.89	10.09

Total	2,796.44	2,718.62	2.78

Table C.5.: *ArmVirtQemu (ARM, GCC5) PE-to-UE Size Comparison.*

Module	**Size** [KiB]		**Saving** [%]
	PE	**UE**	
WatchdogTimer	1.38	0.70	49.43
DpcDxe	1.31	0.71	45.83
HighMemDxe	1.31	0.74	43.45
EnglishDxe	1.75	1.07	38.84
CapsuleRuntimeDxe	1.50	0.93	38.02
ResetSystemRuntimeDxe	2.12	1.52	28.68
SecurityStubDxe	2.62	1.99	24.11
VirtioPciDeviceDxe	3.00	2.37	21.09
ReportStatusCodeRouterRuntimeDxe	2.94	2.36	19.68
MetronomeDxe	2.94	2.36	19.68
MonotonicCounterRuntimeDxe	3.50	2.82	19.42
CpuPei	3.38	2.79	17.36
SerialDxe	3.69	3.06	16.95
VirtioRngDxe	3.94	3.30	16.07
PlatformHasAcpiDtDxe	3.75	3.15	16.04
ArmTimerDxe	4.00	3.41	14.65
ArmPciCpuIo2Dxe	4.44	3.82	13.91
VirtioFdtDxe	4.69	4.07	13.17
SmbiosPlatformDxe	4.81	4.18	13.15
ConPlatformDxe	4.75	4.19	11.84
FdtClientDxe	4.75	4.19	11.84
MemoryInit	5.69	5.05	11.26
QemuKernelLoaderFsDxe	6.00	5.34	11.07
RuntimeDxe	6.00	5.37	10.55
RealTimeClock	5.56	4.98	10.39
VirtioScsiDxe	5.44	4.88	10.20
PlatformPei	5.25	4.73	9.82
VirtioBlkDxe	6.19	5.60	9.47
ArmGicDxe	6.12	5.56	9.18
ScsiBus	6.81	6.20	9.06
Virtio10	7.69	6.99	9.04
DiskIoDxe	7.31	6.67	8.76
ArpDxe	8.12	7.45	8.37
PciHotPlugInitDxe	7.62	7.02	7.99
QemuRamfbDxe	7.00	6.47	7.59
SmbiosDxe	8.19	7.58	7.44
FaultTolerantWriteDxe	8.12	7.52	7.40
VirtioNetDxe	8.12	7.53	7.31

VirtNorFlashDxe	8.75	8.13	7.05
UsbMassStorageDxe	10.00	9.37	6.33
UsbBusDxe	10.88	10.23	5.96
PartitionDxe	12.25	11.52	5.93
PcdDxe	9.44	8.88	5.88
ArmPlatformPrePeiCore	11.44	10.77	5.81
UsbKbDxe	10.19	9.60	5.75
TerminalDxe	12.19	11.51	5.58
UdfDxe	11.88	11.25	5.26
GraphicsConsoleDxe	12.25	11.62	5.17
VirtioGpuDxe	12.62	11.99	5.01
PlatformDxe	13.31	12.66	4.87
UhciDxe	11.25	10.71	4.79
EhciDxe	13.50	12.88	4.57
DxeIpl	12.56	11.99	4.54
DriverHealthManagerDxe	13.69	13.09	4.34
PciHostBridgeDxe	16.06	15.38	4.23
LogoDxe	12.75	12.21	4.23
ConSplitterDxe	14.88	14.25	4.20
VlanConfigDxe	15.62	15.00	4.00
Fat	18.19	17.52	3.69
VirtioFsDxe	16.06	15.48	3.65
LinuxInitrdDynamicShellCommand	17.50	16.88	3.57
MnpDxe	18.94	18.30	3.38
NvmExpressDxe	19.56	18.91	3.31
PeiCore	20.62	19.95	3.26
XhciDxe	19.75	19.15	3.05
Mtftp4Dxe	20.06	19.45	3.04
Dhcp4Dxe	20.44	19.83	2.98
Udp4Dxe	18.88	18.31	2.98
RamDiskDxe	23.94	23.23	2.94
VariableRuntimeDxe	23.69	23.02	2.80
ScsiDisk	23.81	23.20	2.56
PciBusDxe	28.88	28.16	2.49
tftpDynamicCommand	24.50	23.89	2.49
DevicePathDxe	28.62	27.97	2.29
TcpDxe	31.56	30.86	2.23
httpDynamicCommand	30.19	29.55	2.10
Ip4Dxe	39.69	38.99	1.75
SetupBrowser	44.56	43.82	1.67
UefiPxeBcDxe	39.69	39.03	1.65
DisplayEngine	44.38	43.66	1.62
BdsDxe	43.81	43.15	1.52
HiiDatabase	54.00	53.28	1.33
DxeCore	74.12	73.40	0.98
UiApp	81.00	80.31	0.85

Shell	675.56	674.38	0.17
Average	22.70	22.06	9.60
Total	1,929.31	1,875.41	2.79

Table C.6.: *ArmVirtQemu (ARM, CLANGDWARF) PE-to-UE Size Comparison.*

Module	**Size** [KiB]		**Saving** [%]
	PE	**UE**	
WatchdogTimer	1.31	0.65	50.60
HighMemDxe	1.31	0.73	44.05
CapsuleRuntimeDxe	1.56	0.92	41.00
DpcDxe	1.56	0.98	37.50
EnglishDxe	2.12	1.45	31.99
ResetSystemRuntimeDxe	2.38	1.74	26.64
SecurityStubDxe	2.56	1.95	23.78
ReportStatusCodeRouterRuntimeDxe	3.25	2.60	19.95
MetronomeDxe	2.94	2.37	19.41
MonotonicCounterRuntimeDxe	3.38	2.73	18.98
CpuPei	3.50	2.84	18.75
VirtioPciDeviceDxe	3.31	2.71	18.16
PlatformHasAcpiDtDxe	3.69	3.06	16.95
VirtioRngDxe	4.25	3.59	15.44
SerialDxe	3.62	3.07	15.30
ArmTimerDxe	4.38	3.80	13.21
VirtioFdtDxe	4.88	4.26	12.66
SmbiosPlatformDxe	5.31	4.64	12.65
RealTimeClock	5.31	4.66	12.35
ArmPciCpuIo2Dxe	4.88	4.33	11.22
ConPlatformDxe	5.31	4.73	11.03
FdtClientDxe	5.88	5.24	10.77
VirtioScsiDxe	6.06	5.45	10.05
PlatformPei	5.75	5.21	9.38
QemuKernelLoaderFsDxe	6.62	6.02	9.08
VirtioBlkDxe	6.56	5.97	9.05
ArmGicDxe	6.62	6.05	8.61
ScsiBus	7.00	6.45	7.92
MemoryInit	7.94	7.34	7.58
DiskIoDxe	8.25	7.63	7.48
RuntimeDxe	7.88	7.29	7.44
QemuRamfbDxe	7.81	7.24	7.30
PciHotPlugInitDxe	8.56	7.95	7.12
Virtio10	9.62	8.95	6.98
VirtioNetDxe	8.69	8.09	6.83
ArpDxe	9.88	9.22	6.65

VirtNorFlashDxe	10.88	10.20	6.25
FaultTolerantWriteDxe	11.94	11.27	5.56
GraphicsConsoleDxe	13.00	12.34	5.11
PartitionDxe	13.50	12.82	5.03
UsbMassStorageDxe	12.19	11.58	5.00
PcdDxe	12.25	11.65	4.91
TerminalDxe	14.94	14.24	4.65
PlatformDxe	15.06	14.39	4.46
UsbKbDxe	14.00	13.38	4.41
UdfDxe	14.00	13.39	4.35
LogoDxe	12.75	12.20	4.35
VirtioGpuDxe	14.25	13.64	4.28
DriverHealthManagerDxe	15.94	15.27	4.17
SmbiosDxe	16.00	15.34	4.10
DxeIpl	15.44	14.84	3.85
UhciDxe	16.88	16.26	3.66
VlanConfigDxe	17.81	17.16	3.64
ConSplitterDxe	18.75	18.09	3.54
UsbBusDxe	16.31	15.76	3.40
ArmPlatformPrePeiCore	19.50	18.86	3.29
LinuxInitrdDynamicShellCommand	18.12	17.56	3.10
Udp4Dxe	23.44	22.71	3.10
NvmExpressDxe	22.38	21.69	3.07
VirtioFsDxe	20.06	19.46	3.00
PciHostBridgeDxe	24.25	23.57	2.80
EhciDxe	23.25	22.60	2.79
Dhcp4Dxe	24.56	23.89	2.74
MnpDxe	23.50	22.89	2.59
tftpDynamicCommand	27.50	26.80	2.56
Fat	25.88	25.22	2.54
ScsiDisk	27.81	27.12	2.50
Mtftp4Dxe	25.75	25.12	2.46
RamDiskDxe	27.25	26.65	2.21
httpDynamicCommand	35.31	34.59	2.04
PeiCore	35.38	34.68	1.97
VariableRuntimeDxe	34.94	34.29	1.86
XhciDxe	34.50	33.89	1.77
DevicePathDxe	44.00	43.24	1.72
TcpDxe	44.12	43.40	1.65
PciBusDxe	40.38	39.72	1.63
Ip4Dxe	55.19	54.44	1.36
UefiPxeBcDxe	48.62	47.98	1.33
SetupBrowser	64.69	63.90	1.22
BdsDxe	61.50	60.76	1.21
DisplayEngine	59.06	58.36	1.19
HiiDatabase	67.81	67.02	1.16

DxeCore	88.06	87.27	0.90
UiApp	111.75	110.97	0.70
Shell	778.69	777.34	0.17
Average	28.20	27.55	8.70
Total	2,397.06	2,341.73	2.31

Table C.7.: *ArmVirtQemu (AARCH64, GCC5) PE-to-UE Size Comparison.*

Module	**Size** [KiB]		**Saving** [%]
	PE	**UE**	
MetronomeDxe	1.06	0.48	54.41
WatchdogTimer	1.44	0.81	43.48
DpcDxe	1.75	1.14	34.82
EnglishDxe	2.19	1.46	33.21
HighMemDxe	1.50	1.02	32.29
MonotonicCounterRuntimeDxe	1.81	1.24	31.47
CapsuleRuntimeDxe	1.69	1.16	31.02
CpuPei	1.81	1.35	25.43
ArmTimerDxe	2.62	1.96	25.30
ArmPciCpuIo2Dxe	2.94	2.21	24.73
ResetSystemRuntimeDxe	2.94	2.23	23.94
PlatformHasAcpiDtDxe	3.00	2.30	23.44
SerialDxe	3.81	3.05	19.88
PlatformPei	4.56	3.81	16.44
BootGraphicsResourceTableDxe	4.00	3.35	16.21
SecurityStubDxe	4.06	3.42	15.77
VirtioFdtDxe	4.50	3.80	15.45
SmbiosPlatformDxe	4.62	3.93	15.03
RealTimeClock	4.00	3.42	14.45
ReportStatusCodeRouterRuntimeDxe	4.19	3.60	13.99
VirtioPciDeviceDxe	4.88	4.20	13.78
RuntimeDxe	4.44	3.83	13.73
QemuKernelLoaderFsDxe	5.94	5.23	11.97
VirtioRngDxe	5.69	5.02	11.81
VirtioBlkDxe	5.81	5.16	11.16
ArmGicDxe	6.56	5.88	10.36
QemuRamfbDxe	7.25	6.52	10.13
ScsiBus	7.56	6.82	9.81
ConPlatformDxe	6.94	6.30	9.23
VirtNorFlashDxe	7.56	6.87	9.19
FdtClientDxe	7.94	7.22	9.06
DiskIoDxe	7.81	7.12	8.80
QemuFwCfgAcpiPlatform	9.31	8.53	8.39
VirtioScsiDxe	7.88	7.23	8.13

PciHotPlugInitDxe	8.25	7.59	7.95
SmbiosDxe	9.88	9.16	7.28
MemoryInit	7.25	6.75	6.90
PcdDxe	12.25	11.52	5.99
UsbMassStorageDxe	12.44	11.70	5.90
VirtioNetDxe	12.00	11.30	5.79
Virtio10	13.44	12.68	5.64
ArpDxe	13.06	12.38	5.20
PartitionDxe	13.44	12.74	5.17
LogoDxe	13.06	12.43	4.84
UsbKbDxe	14.62	13.95	4.59
UdfDxe	14.44	13.80	4.44
FaultTolerantWriteDxe	11.00	10.52	4.40
DxeIpl	15.62	14.95	4.35
PciHostBridgeDxe	17.31	16.56	4.33
UhciDxe	15.56	14.89	4.32
GraphicsConsoleDxe	16.38	15.68	4.25
ConSplitterDxe	20.50	19.64	4.19
VirtioGpuDxe	15.31	14.67	4.18
DriverHealthManagerDxe	17.00	16.30	4.09
TerminalDxe	18.06	17.34	4.02
UsbBusDxe	18.19	17.47	3.95
VlanConfigDxe	19.62	18.89	3.74
EhciDxe	17.81	17.16	3.64
AcpiTableDxe	18.25	17.59	3.64
PlatformDxe	15.75	15.22	3.37
LinuxInitrdDynamicShellCommand	21.31	20.63	3.19
PeiCore	27.62	26.84	2.86
DevicePathDxe	39.50	38.39	2.81
NvmExpressDxe	25.25	24.55	2.75
Udp4Dxe	25.88	25.16	2.75
Fat	24.94	24.27	2.66
VirtioFsDxe	21.06	20.51	2.63
Mtftp4Dxe	27.31	26.59	2.63
MnpDxe	25.94	25.27	2.56
Dhcp4Dxe	29.62	28.87	2.56
tftpDynamicCommand	30.38	29.63	2.44
XhciDxe	27.81	27.18	2.28
ScsiDisk	31.25	30.55	2.25
RamDiskDxe	29.88	29.22	2.20
httpDynamicCommand	37.69	36.93	2.01
VariableRuntimeDxe	35.12	34.47	1.87
ArmPlatformPrePeiCore	37.75	37.07	1.80
TcpDxe	44.81	44.04	1.73
PciBusDxe	42.50	41.78	1.69
UefiPxeBcDxe	53.88	53.14	1.36

Ip4Dxe	57.56	56.88	1.18
DisplayEngine	58.38	57.69	1.18
SetupBrowser	67.25	66.48	1.14
BdsDxe	67.75	67.02	1.08
DxeCore	105.69	104.59	1.04
HiiDatabase	80.31	79.55	0.94
UiApp	107.50	106.73	0.72
Shell	809.38	804.54	0.60
Average	28.73	27.99	9.52
Total	2,527.88	2,463.16	2.56

Table C.8.: *ArmVirtQemu (AARCH64, CLANGDWARF) PE-to-UE Size Comparison.*

Module	**Size** [KiB]		**Saving** [%]
	PE	**UE**	
MetronomeDxe	1.12	0.49	56.25
WatchdogTimer	1.44	0.80	44.02
HighMemDxe	1.31	0.84	35.71
DpcDxe	1.81	1.20	33.62
CapsuleRuntimeDxe	1.69	1.15	31.94
MonotonicCounterRuntimeDxe	1.94	1.32	31.85
CpuPei	1.56	1.07	31.50
EnglishDxe	2.62	1.86	29.17
ResetSystemRuntimeDxe	2.62	1.98	24.70
PlatformHasAcpiDtDxe	3.12	2.38	23.75
ArmTimerDxe	2.88	2.24	22.01
SerialDxe	3.62	2.91	19.83
ArmPciCpuIo2Dxe	3.62	2.92	19.40
BootGraphicsResourceTableDxe	4.00	3.27	18.16
VirtioFdtDxe	4.44	3.69	16.90
SecurityStubDxe	3.75	3.15	16.04
SmbiosPlatformDxe	4.50	3.78	15.97
ReportStatusCodeRouterRuntimeDxe	4.25	3.62	14.89
PlatformPei	5.19	4.44	14.46
VirtioPciDeviceDxe	4.94	4.23	14.40
RealTimeClock	4.00	3.45	13.87
VirtioRngDxe	5.50	4.84	11.93
VirtioBlkDxe	5.88	5.20	11.44
RuntimeDxe	6.31	5.62	10.89
ConPlatformDxe	7.50	6.77	9.79
ScsiBus	7.62	6.89	9.63
ArmGicDxe	7.25	6.55	9.59
QemuKernelLoaderFsDxe	7.75	7.01	9.58

QemuRamfbDxe	7.88	7.17	8.93
FdtClientDxe	9.06	8.30	8.45
DiskIoDxe	8.81	8.10	8.07
VirtioScsiDxe	8.50	7.82	8.00
QemuFwCfgAcpiPlatform	11.38	10.59	6.94
Virtio10	11.38	10.62	6.59
VirtNorFlashDxe	10.50	9.81	6.55
MemoryInit	9.56	8.95	6.37
PciHotPlugInitDxe	9.38	8.78	6.33
VirtioNetDxe	12.50	11.79	5.69
LogoDxe	12.94	12.27	5.13
UsbMassStorageDxe	14.44	13.72	4.98
PartitionDxe	15.50	14.78	4.64
PcdDxe	16.94	16.16	4.57
GraphicsConsoleDxe	17.56	16.84	4.09
UsbKbDxe	19.00	18.23	4.07
ArpDxe	16.19	15.54	4.01
TerminalDxe	21.44	20.63	3.75
FaultTolerantWriteDxe	14.56	14.02	3.70
UdfDxe	20.19	19.45	3.64
VirtioGpuDxe	17.69	17.05	3.58
SmbiosDxe	20.69	19.96	3.51
DriverHealthManagerDxe	21.31	20.60	3.34
PciHostBridgeDxe	23.50	22.75	3.19
UsbBusDxe	24.31	23.54	3.18
DxeIpl	21.81	21.13	3.12
ConSplitterDxe	26.69	25.87	3.07
AcpiTableDxe	24.00	23.27	3.03
UhciDxe	22.50	21.82	3.02
LinuxInitrdDynamicShellCommand	24.75	24.05	2.84
VlanConfigDxe	26.94	26.20	2.73
NvmExpressDxe	29.06	28.38	2.37
EhciDxe	29.44	28.77	2.28
PlatformDxe	25.06	24.52	2.15
Dhcp4Dxe	36.06	35.29	2.14
MnpDxe	32.25	31.56	2.13
Mtftp4Dxe	37.38	36.62	2.03
Fat	35.19	34.48	2.02
PeiCore	47.12	46.22	1.92
ArmPlatformPrePeiCore	46.69	45.80	1.91
ScsiDisk	38.25	37.52	1.90
tftpDynamicCommand	36.69	35.99	1.90
Udp4Dxe	33.00	32.38	1.89
VirtioFsDxe	29.00	28.45	1.89
RamDiskDxe	38.69	38.02	1.72
httpDynamicCommand	47.19	46.45	1.56

XhciDxe	48.19	47.52	1.39
DevicePathDxe	90.56	89.34	1.35
PciBusDxe	56.38	55.71	1.18
VariableRuntimeDxe	49.75	49.16	1.18
TcpDxe	63.69	62.98	1.10
UefiPxeBcDxe	71.56	70.77	1.10
DxeCore	130.88	129.77	0.85
Ip4Dxe	88.56	87.87	0.79
DisplayEngine	102.19	101.42	0.75
BdsDxe	94.56	93.88	0.73
SetupBrowser	112.88	112.18	0.62
HiiDatabase	148.12	147.33	0.54
UiApp	168.94	168.16	0.46
Shell	1,125.56	1,120.91	0.41
Average	40.08	39.33	9.01
Total	3,527.00	3,460.97	1.87

D. Implementation Fuzz-Testing Code Coverage

This appendix contains the code coverage report data for fuzz testing the UE parsing and conversion code. It was generated by LCOV [132] and is publicly available and reproducible for artifact evaluation [122].

Table D.1.: *UE Parsing Fuzz-Testing Coverage.*

File	**Coverage** [%]		
	Lines	**Functions**	**Branches**
TestUe/Ue.c	100	100	100
TestUefiImage/UefiImage.c	100	100	100
BaseUeImageLib/UeImageLib.c	100	100	100
BaseUefiImageLib/CommonSupport.c	100	100	100
BaseUefiImageLib/UeSupport.c	100	100	100
Total	100	100	100

Table D.2.: *UE Conversion Fuzz-Testing Coverage.*

File	**Coverage** [%]		
	Lines	**Functions**	**Branches**
TestEmit/Emit.c	100	100	100
ImageTool/DynamicBuffer.c	94.4	100	88.9
ImageTool/Image.c	99.1	100	98.9
ImageTool/ImageToolEmit.c	100	100	100
ImageTool/PeScan.c	95.3	100	94.3
ImageTool/UeEmit.c	97.8	100	96.5
ImageTool/UeScan.c	100	100	100
ImageTool/UefiImageScan.c	98.1	100	97.6
Total	98.1	100	97.3

List of Tables

List of Figures

List of Algorithms

List of Source Code Examples

List of Abbreviations

ABI application binary interface 5, 17, 52, 136, 141

ACPI Advanced Configuration and Power Interface 27, 136, 141

ACSL ANSI/ISO C Specification Language 29, 136, 141

AL Afterlife 39, 136, 141

AMD Advanced Micro Devices, Inc 37–40, 136, 141

API application programming interface 11, 26, 66, 69, 70, 88, 136, 141

Apple Apple Inc. 1, 7, 10, 17, 19, 23, 27, 28, 34, 35, 41–43, 89, 90, 136, 141, 149

Arm Arm Limited 2, 26, 52, 81, 91, 136, 141, 143, 144

ASCII American Standard Code for Information Interchange 8, 56, 136, 141

ASLR Address Space Layout Randomization 12, 22, 35, 36, 48, 136, 141, 149

AUDK Acidanthera UEFI Development Kit 22, 27, 28, 77, 78, 80, 81, 85, 87, 136, 141

BDS Boot Device Search 37, 39, 40, 71, 136, 141, 154

BIOS Basic Input/Output System 36, 136, 138, 141

BTI branch target identification 35, 136, 141, 147

C C programming language 4, 6, 7, 9, 10, 28–31, 37, 39, 44, 45, 51, 52, 54–57, 64, 65, 92, 136, 137, 141, 144, 149

CA certification authority 43, 136, 141

CAR cache-as-RAM 37, 44, 136, 141, 150

CET Control-flow Enforcement Technology 35, 136, 141

CFI control flow integrity 136, 141, 147

CI/CD continuous integration and continuous delivery iv, 80, 81, 85, 136, 141

COFF Common Object File Format 10, 21, 136, 139, 141, 151

COW copy-on-write 10, 17, 18, 22, 136, 141

CPU Central Processing Unit 2, 12, 13, 33–38, 40, 44, 45, 48, 54, 55, 88, 136, 141, 145, 147, 149, 150, 152–155

depex dependency expression 40, 88, 89, 136, 141

DXE Driver Execution Environment 38–41, 48, 71, 80, 136, 141, 147, 150, 153, 154

EDK II EFI Development Kit II iii, iv, 16, 25–27, 33, 36, 42, 44, 45, 48, 49, 51, 56, 60, 64, 66, 69, 73, 78, 80, 87, 88, 91, 92, 136, 141, 143, 151

ELF Executable and Linkable Format iii, iv, 10, 11, 17–19, 21–24, 27, 51, 64–66, 73, 77, 136, 141, 151, 152

FFS firmware file system 63, 136, 141

FV firmware volume 136, 141

GCC GNU Compiler Collection 27, 70, 136, 141

GNU GNU's Not Unix! 64, 136, 138, 141

Google Google LLC iv, 136, 141

GOT global offset table 10, 18, 19, 136, 141, 151

HID human interface device 39, 136, 141, 154

HOB Hand-Off Block 87, 136, 141

HRoT Hardware Root of Trust 42, 43, 136, 141, 148, 155

HSM hardware security module 41, 136, 141

IBB Initial Boot Block 42, 136, 141

IBM International Business Machines Corporation 136, 141, 143

IBT indirect branch tracking 35, 136, 141, 145, 147

IBV Independent BIOS Vendor 64, 136, 141

IEEE Institute of Electrical and Electronics Engineers 55, 56, 136, 141

Intel Intel Corporation 1, 2, 24, 26, 33, 35–37, 39, 40, 42, 44, 45, 52, 136, 138, 141, 145, 146, 148–151, 153

Intel ME Intel Management Engine 42, 43, 136, 141

ISA instruction set architecture 2, 12, 17, 26, 48, 49, 52, 54, 58, 87, 136, 141

ISP RAS Ivannikov Institute for System Programming of the RAS iv, 136, 141

JIT just-in-time compilation 34, 136, 141

KASLR Kernel Address Space Layout Randomization 13, 35, 45, 136, 141

KVM Kernel-based Virtual Machine 136, 141, 143, 151

LASS Linear Address Space Separation 33, 136, 141

LTO link-time optimization 70, 71, 90, 136, 141

Mach-O Mach Object File Format iii, 10, 19–23, 27, 28, 51, 64–66, 73, 89, 136, 141, 145, 149, 151

Microsoft Microsoft Corporation 1, 17, 21, 24, 26, 37, 43, 66, 136, 139, 141, 143, 145, 151

MM UEFI PI Management Mode 39–41, 48, 70, 80, 136, 141, 150, 153, 154

MMIO memory-mapped input/output 2, 3, 44, 136, 141, 143, 151

MMU memory management unit 136, 141, 151

MPI-SWS Max Planck Institute for Software Systems 31, 136, 141

MSVC Microsoft Visual C++ 22, 27, 52, 64, 70, 136, 141

NEM no-eviction mode 37, 136, 141

OBB OEM Boot Block 42, 136, 141

OEM original equipment manufacturer 136, 141, 150

OS operating system 136, 141, 151, 152

OVMF Open Virtual Machine Firmware 26, 80–82, 111, 136, 141

PC personal computer 136, 141, 143

PDB program database 65, 66, 136, 141

PE Portable Executable iii, iv, 9, 11, 16, 21, 22, 24–27, 37, 40, 44, 51, 57–61, 64–66, 69, 70, 73, 74, 77, 78, 80–82, 85, 87, 111, 136, 139, 141, 143, 151, 152, 154

PE/COFF Portable Executable and Common Object File Format 21, 27, 136, 141

PEI Pre-EFI Initialization 25, 37–39, 41, 42, 45, 48, 70, 87, 136, 141, 147, 148, 150, 151

PEIM PEI module 40, 44, 45, 59, 136, 141, 145, 150, 151

PIC position-independent code 12, 19, 22, 35, 136, 141, 151

PIE position-independent executable 19, 35, 102, 136, 141

PLT procedure linkage table 8, 11, 18, 19, 136, 141

PPI PEIM-to-PEIM Interface 40, 51, 88, 89, 136, 141

PSP Platform Security Processor 38, 136, 141

PTI page table isolation 33, 136, 141

Qualcomm Qualcomm Technologies, Inc. iv, 37, 136, 141

RAM random access memory 136, 141, 144, 150

Red Hat Red Hat, Inc. iv, 136, 141

RELRO relocation read-only 19, 51, 136, 141

RFC Request for Comments 30, 136, 141

ROP return-oriented programming 7, 8, 136, 141, 153

RoT Root of Trust 136, 141, 144, 153

RT Runtime 39, 40, 45, 59, 66, 80, 136, 141, 150

Rust Rust programming language 4, 29–31, 91, 92, 136, 141

SCO The Santa Cruz Operation, Inc. 17, 136, 141, 146

SDK Software Development Kit 26, 136, 141, 146

SEC Security 37, 48, 87, 136, 141, 148

SMAP Supervisor Mode Access Prevention 33, 136, 141

SMEP Supervisor Mode Execution Prevention 33, 136, 141

SoC System-on-a-Chip 37, 136, 141

SPI Serial Peripheral Interface 24, 40, 42, 44, 81, 136, 141

SRoT Software Root of Trust 37, 136, 141

syscall system call 33, 136, 141

TE Terse Executable iii, 16, 25, 26, 37, 40, 56, 85, 136, 141

TIS Tool Interface Standard 17, 136, 141

TLB translation lookaside buffer 34, 136, 141

TOC/TOU time-of-check to time-of-use 15, 42, 43, 136, 141

TSL Transient System Load 39, 136, 141

UAF use-after-free 136, 141

UE UEFI Executable 51, 60–62, 66, 70–74, 77, 80–82, 90, 91, 105–109, 111, 131, 136, 141

UEFI Unified Extensible Firmware Interface iii, 1, 7, 10, 13–17, 21, 24, 26, 27, 33, 35–37, 39, 40, 43–52, 54–56, 59, 60, 63, 65, 67, 69, 70, 80, 85, 88, 136, 141, 143, 146, 150–152, 154

UEFI HII UEFI Human Interface Infrastructure 44, 64, 65, 136, 141

UEFI PI UEFI Platform Initialization iii, 1, 16, 21, 26, 37–40, 51, 52, 70, 136, 141, 143, 144, 146, 151–154

XIP execute-in-place 25, 27, 44, 60, 61, 73, 81, 87, 136, 141, 151

Glossary

Acidanthera UEFI Development Kit a community fork of EDK II with additional security hardening 22, 136, 137, 141

address space a discrete address range whose mapped data may be context-dependent (e.g. the kernel-space or any user-space process) and may origin from different sources (e.g. main memory or MMIO) 2, 3, 8–11, 13–15, 17–22, 25, 33, 34, 44, 45, 47, 49, 50, 60, 61, 63, 65, 66, 70, 72, 87, 106, 107, 136, 141, 146–151, 155

Address Space Layout Randomization a technique for making the addresses of process components, such as the executable image, call stack, heap memory, or shared libraries, unpredictable by the means of pseudo-randomization 12, 13, 136, 137, 139, 141, 149

Advanced Configuration and Power Interface a specification for data and bytecode tables for an operating system to discover and configure computer hardware components 27, 136, 137, 141

Afterlife the UEFI PI phase during platform resets 39, 136, 137, 141

American Standard Code for Information Interchange de-facto standard for 7-bit string character encoding in computing systems (other standards effectively extend it) 8, 136, 137, 141

application binary interface a specification for conventions regarding anything that concerns binary compatibility between program modules, e.g. sizes and data alignments of data types 5, 136, 137, 141

application programming interface a specification for interfaces used for computer programs to communicate with each other 11, 136, 137, 141

ArmVirtQemu an Arm architecture UEFI PI and UEFI implementation by EDK II for QEMU and KVM 26, 80–82, 111, 136, 141

Authenticode a specification by Microsoft that aggregates various digital signature concepts for the PE format iii, 24, 25, 90, 136, 141

Basic Input/Output System in firmware development, conventionally refers to past firmware implementations originating from ‘IBM PC compatible’ machines 36, 136, 137, 141

big-endian an endianness that lays out bytes from the most to the least significant byte in memory 53, 54, 136, 141

bit-packing a technique for storing data in as few bits as possible without changing its encoding format 56, 136, 141

Boot Device Search the UEFI PI phase to locate the to boot 39, 136, 137, 141, 154

branch target identification an Arm architecture technology to annotate indirect branch targets and to fault when branching to a location without such an annotation 35, 136, 137, 141

buffer overflow a bug of accessing memory beyond its allocated range 6, 21, 136, 141, 144

byte a unit of 8 bits (the definition by C deviates) 1, 2, 9, 20, 22, 50, 52–54, 56, 57, 59, 79, 94–101, 105–109, 136, 141, 143–146, 149

byte-packed the property of forcing a data type's data alignment requirement (for composite types, this includes all of its members) to 1 byte 52–54, 136, 141

cache-as-RAM a technique for (temporarily) using cache memory as if it was system RAM 37, 136, 137, 141

call stack a stack-based data structure, often with hardware support from the processor, to manage local data required by subroutines 4–6, 8, 30, 34–37, 44, 45, 136, 141, 143–145, 152–154

call stack canary typically unpredictable values placed in the current call stack frame, especially near the return address, to detect call stack overflows 34–36, 136, 141

call stack frame a range of the call stack that is dedicated to a single subroutine 4, 5, 8, 34, 35, 136, 141, 144, 152, 154

call stack memory allocation a technique for allocating and free memory at runtime (with the call stack frame layout typically computed at build-time) on the call stack 4, 6, 30, 136, 141

call stack overflow a buffer overflow within the call stack memory 6, 8, 34, 136, 141, 144

call stack smashing an intentional call stack buffer overflow to create new shellcode in the call stack memory 8, 34, 136, 141

certification authority an entity that issues digital certificates and may function as a Root of Trust 41, 136, 137, 141

Chain of Trust starting at the RoT as ground truth, each current entity (e.g. software program) verifies the next, forming a fully trusted chain 40–42, 136, 141

chained image relocation fixups a technique, commonly associated with the Mach Object File Format (Mach-O) format, to manage image relocation fixups as a linked list within the image segment data 20, 59, 60, 136, 141

code-gadget in the field of software exploitation, a sequence of processor instructions already in memory, which is repurposed to enable or perform malicious operations 8, 23, 35, 136, 141, 147, 152

Common Object File Format an object file format specified by Microsoft 21, 136, 137, 139, 141, 151

continuous integration and continuous delivery a joint practice to automatically build, test, and deploy computer programs 80, 136, 137, 141

control flow integrity a technique for enforcing the integrity of the control flow, i.e. there are no branches to unexpected targets 136, 137, 141

Control-flow Enforcement Technology general term for Intel IBT and CPU-assisted shadow call stack support 35, 136, 137, 141

copy-on-write a technique for sharing data, which all consumers may independently modify, by lazily duplicating the source data only when the first write happens 10, 136, 137, 141

cryptographic hash a collision-resistant fixed-size value derived from arbitrary data 15, 23–25, 42, 50, 90, 91, 136, 141

data alignment a constraint that data types must reside at addresses divisible by a specific value 5, 22, 27, 52–54, 57, 61, 63, 74, 105, 136, 141, 143–145, 150

data structure padding a technique for inserting bytes between data structure members or at its end to conform to data alignment requirements 5, 52–54, 136, 141

delta encoding a technique for encoding sequential data as a difference from previous data 136, 141

dependency expression a simple opcode-based language to describe dependencies on PEIMs and UEFI protocols 40, 136, 138, 141

digital certificate a cryptographic key pair that can be used to sign (private key) or authenticate (public key) data 21, 40, 41, 136, 141, 144, 152

digital signature a scheme based on asymmetric cryptography to authenticate arbitrary data iii, 7, 9, 14, 23, 24, 34, 41, 43, 50, 90, 91, 136, 141, 143, 152

double free a bug of attempting to free a previously dynamically allocated memory pointer after it has been freed 7, 136, 141

Driver Execution Environment the fully-featured UEFI PI phase to initialize most of the platform 38, 136, 138, 141

dynamic linker a software, typically part of a firmware or operating system, to perform dynamic linking 1, 11, 13, 89, 136, 141

dynamic linking a technique for loading one or more shared libraries into the address space of an executable at runtime 9, 11, 17, 19, 20, 51, 81, 89, 90, 136, 141, 146, 147, 149, 153

dynamic memory allocation a technique for allocating and free arbitrary amounts of memory with a dynamic lifetime at runtime using heap memory 4–7, 30, 35, 36, 48, 78, 79, 136, 141

EFI Development Kit II the UEFI reference implementation and SDK initially developed by Intel, now managed by TianoCore 16, 136, 138, 141

endianness the byte order of a multi-byte basic data type 53, 136, 141, 143, 149

Executable and Linkable Format an image file format specified by SCO 17, 136, 138, 141

executable file an image file that can be executed by, e.g. a firmware or an operating system iii, iv, 1, 10, 11, 16, 17, 22, 24, 33, 35–37, 43, 49, 60, 63–65, 69, 72, 77, 78, 80, 85, 87, 89, 91, 136, 141, 146–151, 153, 154

execute-in-place a technique for executing a specially crafted executable file from its source storage without performing image file loading or image relocation first 25, 136, 141

fault injection a technique for testing software with intentional faults to observe its error-handling 77–79, 136, 141

file magic number a unique value at the start or end of a file's content that identifies its format 9, 23, 24, 106, 136, 141

firmware a software that is part of a hardware design and often exposes a high-level abstraction for executing hardware operations iii, iv, 1–3, 6, 10, 14, 15, 17, 24, 28, 30, 33, 36–40, 42–45, 48, 49, 51, 59–61, 69–71, 74, 75, 80, 81, 85, 90, 92, 136, 141, 143, 146, 148–151, 153–155

firmware file system a minimal, flat file system for firmware volumes 63, 136, 138, 141

firmware volume a logical unit to store and organize UEFI PI and UEFI modules and accompanying data 40, 42, 136, 138, 141, 146, 150

format string a string utilizing format specifiers, which act like placeholders, to dynamically compose a string 8, 10, 136, 141

forward-edge CFI a technique of annotating targets of indirect branches with CPU architecture specific guards, such as BTI and IBT, so that jumping to code-gadgets is limited to known jump targets 35, 36, 69, 87, 136, 141

fuzz testing a technique for automatically generating inputs for a computer program in order to test its behaviour, e.g. output correctness, internal invariant violations, or exceptions and crashes iv, 28, 29, 77–80, 85, 131, 136, 141

garbage collection a dynamic memory management scheme that periodically scans objects for live references and frees it when it is unreferenced or only referenced in a cycle without live references 28, 136, 141

global offset table an image file section that stores externally-linked, relocatable absolute addresses referenced by read-only image segments 18, 136, 138, 141, 151

Hand-Off Block containers of data storage to pass information from the PEI to the DXE phase 87, 136, 138, 141

hardware security module a device that protects, manages, and performs tasks with secret keys 41, 136, 138, 141

heap memory a memory area dedicated to managing and storing dynamically allocated memory 5, 8, 35, 45, 136, 141, 143, 146

human interface device a class of computer devices which take inputs from humans or output information to humans 39, 136, 138, 141

image a loosely defined term, but in the context of this thesis shall mean the abstract concept of capturing a address space (location, content, access permissions, and potentially other properties) iii, iv, 1, 2, 4, 8–28, 33, 35, 36, 40, 41, 43–51, 56, 57, 59–67, 69–75, 77, 80, 81, 85, 87–91, 102, 106, 109, 136, 141, 143, 145–153

image base address the unique address an image can work from without further modification 9, 11, 13, 20, 22, 35, 44, 65, 66, 136, 141, 148, 151

image entry point the address in an image address space of an executable file to jump to when executing the image 50, 136, 141

image file a loosely defined term, but in the context of this thesis shall mean a file that encodes an image iii, iv, 1, 8–28, 33, 36, 40, 41, 43, 44, 48–51, 56, 57, 59–61, 63–66, 69–75, 77, 80, 81, 85, 87, 88, 90, 91, 136, 141, 146–151

image prelinking a technique for performingdynamic linking at build-time 136, 141

image preloading a technique for performingimage file loading at build-time 44, 136, 141

image relocation a technique for updating all absolute address references of an image from its image base address to a new address 9, 11–13, 17–22, 44–51, 59–62, 66, 70, 72–74, 81, 87–91, 136, 141, 145, 146, 148, 152

image segment the unit of an image file used by an image file loader that composes an image address space 9–12, 14, 17–22, 25, 27, 28, 44, 50, 57, 60, 61, 64–66, 70, 72, 74, 81, 89, 90, 136, 141, 145, 147, 148, 152, 153

image symbol a globally-unique name tied to a value, typically an image address to a function or global variable 11, 50, 51, 65, 66, 90, 102, 136, 141, 152

image file header a data structure with metadata and all information required to parse an image file 8, 9, 13, 14, 24–26, 61, 65, 66, 81, 87, 90, 136, 141

image file linker a software, typically part of a compiler toolchain, to perform static linking. iv, 10–12, 64, 136, 141, 148, 150

image file loader a software, typically part of a firmware or operating system, to load and relocate shared libraries or executable files iii, iv, 1, 13, 23, 26, 27, 40, 41, 50, 66, 69, 70, 72, 91, 136, 141, 148

image file loading the process of extracting the image address space from an image file 9, 10, 13–15, 25, 27, 36, 44, 51, 60, 69, 72, 136, 141, 146, 147

image file section the unit of an image file used by an image file linker that composes an image segment 9–14, 18, 21, 23, 25–28, 51, 64, 90, 91, 136, 141, 147

image relocation fixup an instruction to the image file loader how to fix up an absolute address reference to the image address space 11, 13, 17–22, 45–51, 59, 60, 62, 72, 74, 87–91, 136, 141, 145

indirect branch tracking an Intel architecture technology to annotate indirect branch targets and to fault when branching to a location without such an annotation 35, 136, 138, 141

Initial Boot Block code, typically SEC and PEI, on the firmware storage that is cryptographically authenticated by Intel Boot Guard 42, 136, 138, 141

instruction set architecture an abstract definition of processor instructions, registers, platform features, etc., that is implemented by a microarchitecture 2, 136, 138, 141

Intel Boot Guard a HRoT that verifies the 42, 136, 141, 148, 150

Intel Firmware Support Package 136, 141

just-in-time compilation a technique for compiling code, commonly a form of intermediate and abstract assembly-like language, just in time for its execution 34, 136, 139, 141

Kernel Address Space Layout Randomization ASLR in the kernel-space, using pseudo-randomization for the addresses of the kernel and its extensions 13, 136, 139, 141

Kernel Collection a Mach-O container format for a unified image of the operating system kernel and kernel extensions specified by Apple 89, 90, 136, 141

Kernel-based Virtual Machine a hypervisor that uses hardware virtualization for the Linux kernel 136, 139, 141

kernel-space a CPU organization state for the operating system kernel that usually consists of a high-privilege CPU mode and a page table that maps most of the operating main memory 6, 7, 23, 33, 34, 39, 90, 136, 141, 143, 149–151, 153, 154

lazy dynamic linking a technique for performing dynamic linking only on demand, e.g. when calling an externally imported function 136, 141

libc the C standard library for hosted environments 8, 11, 136, 141, 152

library a collection of functions and global variables that can be consumed by other libraries and executable files by linking iii, 1, 4, 8, 11, 12, 17, 19, 22, 23, 35, 43, 69–72, 74, 75, 77, 88, 90, 136, 141, 143, 146, 148, 149, 151, 153

Linear Address Space Separation a feature of Intel 64 architecture CPUs to prohibit user-space accesses to kernel-space memory and vice-versa by segmenting the process address space into a low half (where user-space memory resides) and a high half (where kernel-space memory resides) 33, 136, 139, 141

linking a technique for enforcing object files and libraries with each other or executable files 9, 11, 13, 17, 19, 20, 27, 51, 81, 88–90, 136, 141, 146–149, 153

link-time optimization a technique for moving code generation to the link-time, which allows code optimization between code compilation units 70, 136, 139, 141

little-endian an endianness that lays out bytes from the least to the most significant byte in memory 53, 54, 136, 141

Mach Object File Format an image file format specified by Apple 19, 136, 139, 141, 145

machine word traditionally, the natural data type of a processor 52, 136, 141

Measured Boot a technique for collecting critical information about the boot environment (e.g. hardware information or firmware configuration) and have a trusted remote attestation entity validate it 14, 39, 43, 136, 141

memory contention a situation of conflict over reserving and accessing main memory 3, 136, 141

memory page a memory management unit, usually at least 4 KiB in size, which can be managed in terms of virtual memory mapping and memory permissions 3, 6, 15, 18–20, 22, 23, 25, 34, 36, 48, 57, 58, 61, 63, 66, 72, 87, 136, 141, 150, 151, 153

memory permissions a restriction on which memory operations can be performed, like reading, writing, and executing 9, 10, 18, 23, 25, 28, 34, 36, 50, 61, 63, 70, 72, 88, 136, 141

memory privilege a restriction on which CPU mode can perform memory operations, e.g. to restrict certain memory to the kernel-space 33, 34, 136, 141

memory swapping sometimes called paging, a technique for temporarily offloading memory page onto secondary memory 3, 15, 20, 136, 141

memory-mapped input/output a technique for mapping device control registers into the CPU address space 2, 136, 139, 141

microcode a translation layer that transforms high-level instructions of a processor into low-level circuit-level operations 36, 37, 136, 141

module dispatcher a phase-specific program that locates PEIMs, UEFI-related drivers, or UEFI PI Management Mode (MM) drivers and starts them with regard to their dependencies 38, 40, 41, 44, 48, 64, 65, 71, 89, 136, 141

natural data alignment a data alignment requirement on basic data types that is equal to their size 5, 52, 53, 105, 136, 141

no-eviction mode on Intel platforms, the implementation of CAR, which disables cache line eviction to persist its content 37, 136, 139, 141

non-volatile RAM a feature of UEFI-based platforms to persistently store variables, such as for the firmware configuration 40, 136, 141

`NULL`-pointer dereference a bug of attempting to dereference a `NULL`-pointer 6, 136, 141

object file an intermediate image file generated by a compiler from compilation units, which can be combined with other such into an executable file by an image file linker 11, 18, 88, 89, 91, 136, 141, 145, 149, 153

OEM Boot Block firmware volumes, typically DXE, RT, and MM, shadowed to memory that are cryptographically authenticated by PEI as part of Intel Boot Guard 42, 136, 139, 141

ones' complement a signed integer representation that encodes negatives values as their bitwise inverse 55, 136, 141

Open Virtual Machine Firmware an Intel architecture UEFI PI and UEFI implementation by EDK II for QEMU and KVM 26, 136, 139, 141

OS loader a software that locates, loads, and hands over control to the operating system kernel 36, 37, 39–41, 90, 136, 141, 144, 154

page fault an exception raised by a MMU when accessing a memory page fails (e.g. the memory page is unmapped or the processor is in an underprivileged state) 3, 18, 136, 141

page table a data structure to map virtual memory memory pages to physical memory memory pages 33, 34, 136, 140, 141, 149, 151, 155

page table isolation a technique for mapping only the minimal amount of kernel-space memory pages into user-space address spaces 33, 136, 140, 141

Portable Executable and Common Object File Format an umbrella term for PE and COFF due to their shared technical foundations 21, 136, 139, 141

PEI module an executable module designated to run during the UEFI PI PEI phase 40, 136, 139, 141

PEIM shadowing a technique for transferring an XIP image from the firmware storage to the main memory at runtime 44, 45, 136, 141

PEIM-to-PEIM Interface a globally-uniquely identified shared data structure that is discoverable via PEI services 40, 136, 140, 141

physical memory a concept to directly address main memory and MMIO 2, 3, 136, 141, 151

Portable Executable an executable file format specified by Microsoft 21, 25, 136, 139, 141, 151

position-independent code a concept of ELF and Mach-O image files to utilize relative addressing and a GOT to emit shared libraries that can be loaded at an arbitrary image base addresss 12, 136, 140, 141

position-independent executable similar to PIC, but for executable files 19, 136, 140, 141

Pre-EFI Initialization the reduced-functionality UEFI PI phase to initialize the main memory 37, 136, 139, 141

procedure linkage table function call stubs analogous to global offset table 18, 136, 140, 141

program database a database of image debug information, e.g. image symbols, used in combination with a PE file 65, 136, 139, 141

pseudo-randomization a technique for producing values that are close enough to statistically random values by deterministic means 12, 22, 136, 141, 143, 149

readers-writer lock a lock data structure that allows either an arbitrary number of readers or exactly one writer 30, 136, 141

reference counting a dynamic memory management scheme that automatically tracks the number of live references to an object and frees it when it is unreferenced 28, 136, 141

relative addressing a technique for encoding addresses relative to the current CPU instruction 2, 12, 13, 18, 136, 141, 151

relocation read-only a technique for enforcing write protection on image segments related to image relocation in ELF files 19, 136, 140, 141

reset vector the location from which the CPU fetches the first instruction 2, 42, 136, 141, 153

return address the address of the instruction following the current subroutine call, stored in the current call stack frame 5, 6, 8, 34, 35, 136, 141, 144, 152, 153

return-oriented programming a generalization of a return-to-libc attack, involving arbitrary code-gadget chain to eventually achieve arbitrary code execution 7, 136, 140, 141

return-to-libc a technique for exploiting programs by corrupting the call stack to overwrite the return address with a libc address or another code-gadget, typically to invoke shell commands 8, 136, 141, 152

root digital certificate a self-signed digital certificate that is axiomatically trusted by an authority-based system design 40, 41, 136, 141

Root of Trust an entity in a secure system design that is axiomatically trusted 37, 40, 42, 136, 138, 140, 141, 144

Runtime the UEFI PI phase during the execution of an operating system 39, 136, 140, 141

runtime image relocation a technique for performing image relocation on an image during its runtime when handing control over from UEFI to the operating system kernel 13, 45–47, 50, 136, 141

Secure Boot a technique for authenticating UEFI operating system (OS) loaders and third-party drivers using digital signatures 7, 14, 23, 24, 39, 43, 136, 141

Security the first UEFI PI phase that is invoked after power events, such as the reset vector, which is the software RoT 37, 136, 140, 141

Serial Peripheral Interface a synchronous serial communication interface often found in personal computers and embedded systems, e.g. to connect the firmware storage to the chipset or CPU 24, 136, 140, 141

shadow call stack a stack structure, sometimes managed internally by the CPU, that keeps a history of return addresses to detect ROP attacks 35, 136, 141, 145

shared library a library that is consumable by dynamic linking 4, 8, 11, 12, 17, 19, 22, 35, 43, 90, 136, 141, 143, 146, 148, 151

shellcode an attacker-controlled code payload that may start a privileged command shell or perform other malicious actions 8, 136, 141, 144

side-channel a source of metadata that allows inferring secret information, e.g. determining whether a kernel-space memory page is present by measuring memory access timing from user-space 22, 33, 136, 141

sign and magnitude a signed integer representation that encodes negative values by setting the most significant bit 55, 136, 141

Software Development Kit a collection of tools and libraries that aid developing for a specific software platform or product 26, 136, 140, 141

Standalone MM a MM implementation that is independent of DXE Services and restricts drivers to MM Services 39, 136, 141

static library a library that is consumable by static linking 11, 88, 136, 141

static linking a technique for combining multiple object files into a library or executable file at build-time 9, 11, 13, 17, 27, 51, 88–90, 136, 141, 148, 153

static memory allocation a technique for allocating fixed amounts of memory at build-time, commonly utilizing a data image segment 3, 4, 8, 23, 136, 141

Supervisor Mode Access Prevention a feature of Intel architecture CPUs to allow read and write accesses to user-space memory only when explicitly allowed 33, 136, 140, 141

Supervisor Mode Execution Prevention a feature of Intel architecture CPUs to allow execution of user-space memory only when explicitly allowed 33, 136, 140, 141

system call a command by a user-space to be executed in the kernel-space 33, 136, 140, 141

tail-recursion elimination a technique for optimizing a recursive function call at the function's end to re-use the previous step's call stack frame 4, 136, 141

Terse Executable a stripped variant of PE defined by the UEFI PI specification 25, 136, 140, 141

time-of-check to time-of-use a software vulnerability that allows for data to be invalidated after authentication and before processing is complete 15, 136, 141

Traditional MM a MM implementation that is dependent on DXE Services and exposes them to its drivers 39, 136, 141

Transient System Load the UEFI PI transitional phase between the OS loader execution and handing off to the operating system kernel 39, 136, 141

translation lookaside buffer a cache of the recently translated virtual address to physical address mappings 34, 136, 140, 141

trusted timestamp a timestamp that has been digitally signed by a trusted authority to be authentic 41, 136, 141

two's complement a signed integer representation that encodes negative values as their positive complement regarding 2^N 54, 55, 136, 141

UEFI Boot Manager the UEFI counterpart to Boot Device Search (BDS), which handles HID and boot policies 37, 39, 136, 141

UEFI Driver Model the UEFI design to support device drivers to probe and attach to hardware devices 39, 136, 141

UEFI Executable an executable file format draft proposed by this thesis 51, 105, 136, 141

UEFI Human Interface Infrastructure an UEFI concept to manage user input and output, including forms, resources, and localization 44, 136, 141

UEFI Platform Initialization a firmware interface specification by the UEFI Forum concerned with hardware initialization for the IA32, X64, ARM, AArch64, RISC-V, and LoongArch CPU architectures 37, 136, 141

UEFI protocol a globally-uniquely identified shared data structure that is discoverable via UEFI boot services 39, 40, 51, 88, 89, 136, 141, 145

UEFI PI Management Mode a high-privilege CPU mode that exceeds the kernel-space 39, 136, 139, 141, 150

Unified Extensible Firmware Interface a firmware interface specification by the UEFI Forum concerned with pre-boot drivers and operating system booting for the IA32, X64, ARM, AArch64, RISC-V, and LoongArch CPU architectures 37, 136, 141

use-after-free a bug of accessing a previously dynamically allocated memory pointer after it has been freed 6, 7, 30, 35, 136, 141

use-after-scope a bug of accessing previously automatically allocated memory after its scope has ended 6, 136, 141

user-space a CPU organization state for user-invoked processes that usually consists of a low-privilege CPU mode and a limited, process-specific page table 7, 8, 11, 23, 30, 33, 43, 90, 136, 141, 143, 149, 151, 153

Verified Boot a technique for authenticating the entire platform firmware boot process, typically with using a HRoT 42, 136, 141

virtual memory a concept to dynamically manage an address space that can be aggregated from, e.g. physical main memory and permanent storage devices 3, 10, 13–15, 24, 33–35, 45, 51, 59, 136, 141, 150, 151

W^X a technique for prohibiting memory to be writable and executable at the same time (pronounced ‘W XOR X’) 34, 36, 63, 64, 136, 141

Bibliography

[1] Avi Silberschatz, Peter Baer Galvin and Greg Gagne. *Operating System Concepts*. Tenth. Hoboken, NJ, USA: John Wiley & Sons, Apr. 2018. ISBN: 978-1-118-06333-0. URL: https://os-book.com (visited on 02/07/2023).

[2] UEFI Forum, Inc. *UEFI Platform Initialization (PI) Specification*. Version 1.7 Errata A. Apr. 2020. URL: https://uefi.org/sites/default/files/resources/PI_Spec_1_7_A_final_May1.pdf (visited on 29/10/2022).

[3] UEFI Forum, Inc. *Unified Extensible Firmware Interface (UEFI) Specification*. Release 2.10. Aug. 2022. URL: https://uefi.org/sites/default/files/resources/UEFI_Spec_2_10_Aug29.pdf (visited on 29/10/2022).

[4] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. Online. Version 1.2. May 1995. URL: https://refspecs.linuxbase.org/elf/elf.pdf (visited on 29/10/2022).

[5] Apple Inc. *OS X ABI Mach-O File Format Reference*. Online. Cupertino, CA, USA, Feb. 2009. URL: https://web.archive.org/web/20100114012151/http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html (visited on 29/10/2022).

[6] Microsoft Corporation et al. *PE Format*. Online. June 2022. URL: https://github.com/MicrosoftDocs/win32/blob/33f79279f16b9f4608b3bb1afd0ce6e27280d70b/desktop-src/Debug/pe-format.md (visited on 29/10/2022).

[7] Microsoft Corporation. *Windows 10 Minimum Hardware Requirements*. Online. Redmond, WA, USA, Dec. 2019. URL: https://download.microsoft.com/download/c/1/5/c150e1ca-4a55-4a7e-94c5-bfc8c2e785c5/Windows%2010%20Minimum%20Hardware%20Requirements.pdf (visited on 04/11/2022).

[8] Apple Inc. *XNU*. Online. Version 8796.101.5. Cupertino, CA, USA, June 2023. URL: https://github.com/apple-oss-distributions/xnu/tree/aca3beaa3dfbd42498b42c5e5ce20a938e6554e5 (visited on 12/07/2023).

[9] Alexander W. Dent. *Secure Boot and Image Authentication. Technical Overview.* Online. Version 2.0. San Diego, CA, US: Qualcomm Technologies, Inc., 2019. URL: https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/secure-boot-and-image-authentication-version_final.pdf (visited on 04/11/2022).

[10] Linus Torvalds et al. *Linux.* Online. Version 6.4. June 2023. URL: https://github.com/torvalds/linux/tree/6995e2de6891c724bfeb2db33d7b87775f913ad1 (visited on 13/07/2023).

[11] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4.* Online. Order Number: 325462-079US. Santa Clara, CA, USA, Apr. 2022. URL: https://cdrdv2.intel.com/v1/dl/getContent/671200 (visited on 10/12/2022).

[12] Arm Limited. *Arm® Architecture Reference Manual. for A-profile architecture.* Online. Version J.a. Cambridge, England, UK. URL: https://developer.arm.com/documentation/ddi0487/ja/ (visited on 25/06/2023).

[13] Chris Down. *In defence of swap: common misconceptions.* Online. Jan. 2018. URL: https://chrisdown.name/2018/01/02/in-defence-of-swap.html (visited on 25/06/2023).

[14] ISO. *ISO/IEC 9899:2018 Information technology — Programming languages — C.* Fourth. Geneva, Switzerland: International Organization for Standardization, June 2018. URL: https://www.iso.org/standard/74528.html (visited on 29/10/2022).

[15] The Rust Project Developers. *The Rust Reference.* Version 1.70.0. Apr. 2023. URL: https://doc.rust-lang.org/1.70.0/reference/ (visited on 24/06/2023).

[16] Owen Kaser, C. R. Ramakrishnan and Shaunak Pawagi. 'On the Conversion of Indirect to Direct Recursion'. In: *ACM Lett. Program. Lang. Syst.* 2.1–4 (Mar. 1993), pp. 151–164. ISSN: 1057-4514. DOI: 10.1145/176454.176510.

[17] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger and Mark Mitchell. *System V Application Binary Interface. AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models).* Version 1.0. May 2023. URL: https://gitlab.com/x86-psABIs/x86-64-ABI/-/tree/ca48acd297294172349d4057fb0f04016c5733bd (visited on 01/07/2023).

[18] MISRA. *MISRA-C:2012. Guidelines for the use of the C language critical systems.* Third. Nuneaton, Warks, UK: HORIBA MIRA Limited, Feb. 2019.

[19] Christoph Lameter. *SLUB: The unqueued slab allocator V6.* Online. Silicon Graphics International, Mar. 2007. URL: https://lwn.net/Articles/229096/ (visited on 29/05/2023).

[20] I. Puaut. 'Real-time performance of dynamic memory allocation algorithms'. In: *Proceedings 14th Euromicro Conference on Real-Time Systems. Euromicro RTS 2002*. 2002, pp. 41–49. DOI: 10.1109/EMRTS.2002.1019184.

[21] TianoCore et al. *EFI Development Kit II*. Online. Commit 93629f2. Nov. 2022. URL: https://github.com/tianocore/edk2/tree/93629f2c7cf05ebc0f458eacc42a33147096f9d1 (visited on 16/11/2022).

[22] Tony Hoare. *QCon London 2009. Null References: The Billion Dollar Mistake*. Online. Presentation abstract. Mar. 2009. URL: https://qconlondon.com/london-2009/qconlondon.com/london-2009/presentation/Null%2bReferences_%2bThe%2bBillion%2bDollar%2bMistake.html (visited on 07/07/2023).

[23] Kyung-Suk Lhee and Steve J. Chapin. 'Buffer overflow and format string overflow vulnerabilities'. In: *Software: Practice and Experience* 33.5 (2003), pp. 423–460. DOI: https://doi.org/10.1002/spe.515.

[24] Apple Inc. *macOS User Guide. Change security settings on the startup disk of a Mac with Apple silicon*. Online. macOS Ventura 13. 2022. URL: https://support.apple.com/guide/mac-help/change-security-settings-startup-disk-a-mac-mchl768f7291/mac/13.0 (visited on 25/05/2023).

[25] ESET, spol. s r.o. *LoJax. First UEFI rootkit found in the wild, courtesy of the Sednit group*. Online. Bratislava, SK, Sept. 2018. URL: https://www.welivesecurity.com/wp-content/uploads/2018/09/ESET-LoJax.pdf (visited on 11/07/2023).

[26] Martin Smolár. *BlackLotus UEFI bootkit: Myth confirmed*. Online. Bratislava, SK: ESET, spol. s r.o., Mar. 2023. URL: https://www.welivesecurity.com/2023/03/01/blacklotus-uefi-bootkit-myth-confirmed/ (visited on 11/07/2023).

[27] Victor van der Veen, Nitish dutt-Sharma, Lorenzo Cavallaro and Herbert Bos. 'Memory Errors: The Past, the Present, and the Future'. In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Davide Balzarotti, Salvatore J. Stolfo and Marco Cova. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 86–106. DOI: 10.1007/978-3-642-33338-5_5.

[28] Solar Designer. *Getting around non-executable stack (and fix)*. Online. Aug. 1997. URL: https://seclists.org/bugtraq/1997/Aug/63 (visited on 02/07/2023).

[29] Nergal. 'Advanced return-into-lib(c) exploits (PaX case study)'. In: *Phrack Magazine* 11.4 (58 Dec. 2001). Ed. by Phrack Staff. URL: http://phrack.org/issues/58/4.html (visited on 02/07/2023).

[30] Joe Bialek, Ken Johnson, Matt Miller and Tony Chen. *Security Analysis of Memory Tagging.* Online. Microsoft Corporation, Mar. 2020. URL: https://github.com/microsoft/MSRC-Security-Research/blob/4ef90e324189e511638952afd6102a6c59e5af11/papers/2020/Security%20analysis%20of%20memory%20tagging.pdf (visited on 01/07/2023).

[31] Maddie Stone. *The More You Know, The More You Know You Don't Know. A Year in Review of 0-days Used In-the-Wild in 2021.* Online. Google Project Zero, Apr. 2022. URL: https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html (visited on 01/07/2023).

[32] Apple Security Engineering and Architecture (SEAR). *Towards the next generation of XNU memory safety: kalloc_type.* Online. Apple Inc., Oct. 2022. URL: https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/ (visited on 01/07/2023).

[33] John R. Levine. *Linkers and Loaders.* First. Amsterdam, NH, NL: Elsevier, Oct. 1999. ISBN: 978-1-558-60496-4.

[34] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection. For gcc version 13.1.0.* Boston MA, USA: GNU Press, Apr. 2023. URL: https://gcc.gnu.org/onlinedocs/gcc-13.1.0/gcc.pdf (visited on 06/07/2023).

[35] Raymond Chen. *The New One Thing. If relocated DLLs cannot share pages, then doesn't ASLR cause all pages to be non-shared?* Online. Redmond, WA, USA: Microsoft Corporation, Apr. 2016. URL: https://devblogs.microsoft.com/oldnewthing/20160413-00/?p=93301 (visited on 26/02/2023).

[36] Hector Marco-Gisbert and Ismael Ripoll Ripoll. 'Address Space Layout Randomization Next Generation'. In: *Applied Sciences* 9.14 (2019). ISSN: 2076-3417. DOI: 10.3390/app9142928. URL: https://www.mdpi.com/2076-3417/9/14/2928.

[37] Arm Limited. *ELF for the Arm® 64-bit Architecture (AArch64).* Online. Cambridge, England, UK, Apr. 2023. URL: https://github.com/ARM-software/abi-aa/releases/download/2023Q1/aaelf32.pdf (visited on 12/07/2023).

[38] Apple Inc. *Apple File System Reference.* Online. Cupertino, CA, USA, June 2020. URL: https://developer.apple.com/support/downloads/Apple-File-System-Reference.pdf (visited on 11/07/2023).

[39] Ralph C. Merkle. *Method of providing digital signatures.* United States Patent US 4,309,569. United States Patent and Trademark Office, Jan. 1982.

[40] Marvin Häuser and Vitaly Cheptsov. 'Securing the EDK II Image Loader'. In: *2020 Ivannikov Ispras Open Conference (ISPRAS).* Dec. 2020, pp. 16–25. DOI: 10.1109/ISPRAS51486.2020.00010.

[41] Marvin Häuser. *GNU-EFI Issue #27. Malformed PE image generation.* Online. June 2021. URL: `https://sourceforge.net/p/gnu-efi/bugs/27/` (visited on 25/05/2023).

[42] Marvin Häuser. *iPXE Issue #313. Improve ELF to PE conversion.* Online. Apr. 2021. URL: `https://github.com/ipxe/ipxe/pull/313` (visited on 20/01/2023).

[43] The Santa Cruz Operation, Inc. and AT&T. *System V Application Binary Interface.* Edition 4.1. Santa Cruz, WA, USA: The Santa Cruz Operation, Inc., Mar. 1997. URL: `http://www.sco.com/developers/devspecs/gabi41.pdf` (visited on 08/11/2022).

[44] The Santa Cruz Operation, Inc. and AT&T. *System V Application Binary Interface.* Update. Santa Cruz, WA, USA: The Santa Cruz Operation, Inc., Dec. 2012. URL: `http://www.sco.com/developers/gabi/2012-12-31/contents.html` (visited on 08/11/2022).

[45] The Santa Cruz Operation, Inc. and AT&T. *System V Application Binary Interface. Intel386™ Architecture Processor Supplement.* Fourth. Santa Cruz, WA, USA: The Santa Cruz Operation, Inc., Mar. 1997. URL: `http://www.sco.com/developers/devspecs/abi386-4.pdf` (visited on 29/10/2022).

[46] Rahul Chaudhry. *Generic System V Application Binary Interface. Proposal for a new section type SHT_RELR.* Online. Dec. 2017. URL: `https://groups.google.com/g/generic-abi/c/bX460iggiKg` (visited on 06/07/2023).

[47] Steve Chamberlain and Ian Lance Taylor. *The GNU linker.* Ed. by Jeffrey Osier. GNU Binutils. Version 2.40. Free Software Foundation, Inc., Jan. 2023. URL: `https://sourceware.org/binutils/docs-2.40/ld.pdf` (visited on 13/07/2023).

[48] Noah Martin. *How iOS 15 makes your app launch faster.* Online. June 2021. URL: `https://www.emergetools.com/blog/posts/iOS15LaunchTime` (visited on 30/10/2022).

[49] Acidanthera. *Acidanthera UEFI Development Kit.* Online. Commit 6ca1d30. June 2023. URL: `https://github.com/acidanthera/audk/tree/6ca1d30f606125df8f0b3ec35eebce93138bf3c9` (visited on 03/07/2023).

[50] Raymond Chen. *The New One Thing. How important is it nowadays to ensure that all my DLLs have non-conflicting base addresses?* Online. Redmond, WA, USA: Microsoft Corporation, Jan. 2017. URL: `https://devblogs.microsoft.com/oldnewthing/20170120-00/?p=95225` (visited on 26/02/2023).

[51] David Howells, David Woodhouse and Juerg Haefliger. *sign-tool. Sign a module file using the given key.* Online. The Linux Kernel: Red Hat, Inc., Intel Corporation. and Hewlett Packard Enterprise Development LP, Aug. 2022. URL: `https://github.com/torvalds/linux/blob/e01d50cbd6eece456843717a566a34e8b926cf0c/scripts/sign-file.c` (visited on 15/11/2022).

[52] Apple Inc. *TN3126: Inside Code Signing: Hashes.* Online. Cupertino, CA, USA, May 2022. URL: `https://developer.apple.com/documentation/technotes/tn3126-inside-code-signing-hashes#Special-slots` (visited on 13/11/2022).

[53] Apple Inc. *TN2206: macOS Code Signing In Depth.* Online. Cupertino, CA, USA, Aug. 2016. URL: `https://developer.apple.com/library/archive/technotes/tn2206/_index.html` (visited on 13/11/2022).

[54] Acidanthera. *OpenCore bootloader.* Online. Commit dc182df. June 2023. URL: `https://github.com/acidanthera/OpenCorePkg/tree/dc182df42c519b70ff223bdbcd4d6de1455be362` (visited on 03/07/2023).

[55] Microsoft Corporation. *Windows Authenticode Portable Executable Signature Format.* Online. Version 1.0. Redmond, WA, USA, Mar. 2008. URL: `https://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/Authenticode_PE.docx` (visited on 06/11/2022).

[56] Bernardo Quintero. *Distribution of malicious JAR appended to MSI files signed by third parties.* Online. VirusTotal, Jan. 2019. URL: `https://blog.virustotal.com/2019/01/distribution-of-malicious-jar-appended.html` (visited on 07/11/2022).

[57] Tomislav PeriČin. *Breaking the Microsoft Authenticode security model.* Online. ReversingLabs, Oct. 2019. URL: `https://blog.reversinglabs.com/blog/rocking-the-foundations-of-a-trust-based-digital-code-signing-system` (visited on 07/11/2022).

[58] Tomislav PeriČin. *Breaking the Windows Authenticode security model.* Online. ReversingLabs, Oct. 2019. URL: `https://blog.reversinglabs.com/blog/breaking-the-windows-authenticode-security-model` (visited on 07/11/2022).

[59] Tomislav PeriČin. *Breaking the UEFI firmware Authenticode security model.* Online. ReversingLabs, Oct. 2019. URL: `https://blog.reversinglabs.com/blog/breaking-uefi-firmware-authenticode-security-model` (visited on 07/11/2022).

[60] Microsoft Corporation et al. *Project Mu BaseCore.* Online. Commit 347cce4. July 2023. URL: `https://github.com/microsoft/mu_basecore/tree/347cce4949a5073b284d7d73d307797659033ff0` (visited on 14/07/2023).

[61] QEMU. *QEMU. A generic and open source machine emulator and virtualizer.* Online. URL: `https://www.qemu.org` (visited on 03/07/2023).

[62] Michael Brown. *iPXE. iPXE network bootloader*. Online. July 2023. URL: https://github.com/ipxe/ipxe/tree/e17568ad0642490143d0c6b154c874b9b9e285bf (visited on 01/07/2023).

[63] Apple Inc. *cctools*. Online. Version 973.0.1. Cupertino, CA, USA, Oct. 2021. URL: https://opensource.apple.com/source/cctools/cctools-973.0.1/ (visited on 21/11/2022).

[64] Acidanthera. *ocmtoc*. Online. Commit 6f70907. Jan. 2023. URL: https://github.com/acidanthera/ocmtoc/tree/6f709079e1770206d62bc3abdce73a106aa5c9ff (visited on 27/01/2023).

[65] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser and Christian Holler. *The Fuzzing Book*. Commit 5cbdedd. Saarbrücken, SL, Germany: CISPA Helmholtz Center for Information Security, July 2023. URL: https://github.com/uds-se/fuzzingbook/tree/5cbdedd02abee0aab9341afcf518b290217b8309 (visited on 07/07/2023).

[66] LLVM Project. *libFuzzer. a library for coverage-guided fuzz testing*. Online. Mar. 2023. URL: https://releases.llvm.org/16.0.0/docs/LibFuzzer.html (visited on 03/07/2023).

[67] The Clang Team. *Clang 16.0.0 documentation. SanitizerCoverage*. Online. Mar. 2023. URL: https://releases.llvm.org/16.0.0/tools/clang/docs/SanitizerCoverage.html (visited on 06/07/2023).

[68] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao and Jiaguang Sun. 'SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing'. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 61–64. ISBN: 9781450356633. DOI: 10.1145/3183440.3183494. URL: https://doi.org/10.1145/3183440.3183494.

[69] Alexey Vishnyakov, Andrey Fedotov, Daniil Kuts, Alexander Novikov, Darya Parygina, Eli Kobrin, Vlada Logunova, Pavel Belecky and Shamil Kurmangaleev. 'Sydr: Cutting Edge Dynamic Symbolic Execution'. In: *2020 Ivannikov Ispras Open Conference (ISPRAS)*. 2020, pp. 46–54. DOI: 10.1109/ISPRAS51486.2020.00014.

[70] The Clang Team. *Clang 16.0.0 documentation. AddressSanitizer*. Online. Mar. 2023. URL: https://releases.llvm.org/16.0.0/tools/clang/docs/AddressSanitizer.html (visited on 03/07/2023).

[71] Konstantin Serebryany, Derek Bruening, Alexander Potapenko and Dmitriy Vyukov. 'AddressSanitizer: A Fast Address Sanity Checker'. In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, June 2012, pp. 309–318. ISBN: 978-931971-93-5. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[72] The Clang Team. *Clang 16.0.0 documentation. UndefinedBehaviorSanitizer*. Online. Mar. 2023. URL: `https://releases.llvm.org/16.0.0/tools/clang/docs/UndefinedBehaviorSanitizer.html` (visited on 03/07/2023).

[73] Makarius Wenzel, Clemens Ballarin, Stefan Berghofer, Jasmin Blanchette et al. *The Isabelle/Isar Reference Manual.* Online. Oct. 2022. URL: `https://isabelle.in.tum.de/dist/Isabelle2022/doc/isar-ref.pdf` (visited on 08/07/2023).

[74] The Coq Development Team. *The Coq Reference Manual.* Online. Release 8.17.1. June 2023. URL: `https://github.com/coq/coq/releases/download/V8.17.1/coq-8.17.1-reference-manual.pdf` (visited on 08/07/2023).

[75] Morten Heine Sørensen and Pawel Urzyczyn. In: *Lectures on the Curry-Howard isomorphism.* First. Vol. 149. Studies in Logic and the Foundations of Mathematics. Amsterdam, NH, NL: Elsevier, July 2006. ISBN: 978-0-444-52077-7.

[76] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister and Christian Ferdinand. 'CompCert - A Formally Verified Optimizing Compiler'. In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress.* SEE. Toulouse, France, Jan. 2016. URL: `https://inria.hal.science/hal-01238879` (visited on 03/07/2023).

[77] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles and Boris Yakobowski. 'Frama-C'. In: *Software Engineering and Formal Methods.* Ed. by George Eleftherakis, Mike Hinchey and Mike Holcombe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 233–247. ISBN: 978-3-642-33826-7.

[78] Pascal Cuoq, David Delmas, Stéphane Duprat and Victoria Moya Lamiel. 'Fan-C, a Frama-C plug-in for data flow verification'. In: *Embedded Real Time Software and Systems (ERTS2012).* Toulouse, France, Feb. 2012. URL: `https://hal.science/hal-02263407`.

[79] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy and Virgile Prevosto. *ANSI/ISO C Specification Language.* Online. Version 1.19. CEA LIST, Inria, June 2023. URL: `https://frama-c.com/download/acsl-1.19.pdf` (visited on 12/07/2023).

[80] Dana Jansens, Łukasz Anforowicz and Chris Palmer. *Borrowing Trouble: The Difficulties Of A C++ Borrow-Checker.* Online. Sept. 2021. URL: `https://docs.google.com/document/d/e/2PACX-1vSt2VB1zQAJ6JDMaIA9PlmEgBxz2K5Tx6w2JqJNeYCy0gU4aoubdTxlENSKNSrQ2TXqPWcuwtXe6PlO/pub` (visited on 17/07/2023).

[81] Miguel Ojeda. *Linux kernel mailing list. [PATCH 00/13] [RFC] Rust support.* Online. Message ID <20210414184604.23473-1-ojedakernel.org>. Apr. 2021. URL: `https://lore.kernel.org/lkml/20210414184604.23473-1-ojeda@kernel.org/` (visited on 17/07/2023).

[82] Linus Torvalds. *Linux kernel source tree. Merge tag 'rust-v6.1-rc1' of `https://github.com/Rust-for-Linux/linux`.* Online. Commit 8aebac8. Oct. 2022. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8aebac82933ff1a7c8eede18cab11e1115e2062b` (visited on 17/07/2023).

[83] Max Planck Institute for Software Systems. *ERC Project "RustBelt".* Online. Apr. 2021. URL: `https://plv.mpi-sws.org/rustbelt/` (visited on 17/07/2023).

[84] Jack Huey. *Rust Blog. Officially announcing the types team.* Online. Jan. 2023. URL: `https://blog.rust-lang.org/2023/01/20/types-announcement.html` (visited on 17/07/2023).

[85] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher et al. 'Meltdown: Reading Kernel Memory from User Space'. In: *Commun. ACM* 63.6 (May 2020), pp. 46–56. DOI: `10.1145/3357033`.

[86] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice and Stefan Mangard. 'KASLR is Dead: Long Live KASLR'. In: *Engineering Secure Software and Systems.* Ed. by Eric Bodden, Mathias Payer and Elias Athanasopoulos. Cham: Springer International Publishing, 2017, pp. 161–176. DOI: `10.1007/978-3-319-62105-0_11`.

[87] The kernel development community. *The Linux Kernel documentation. Page Table Isolation (PTI).* Online. Version 6.0.0. Oct. 2022. URL: `https://www.kernel.org/doc/html/v6.0/x86/pti.html` (visited on 10/12/2022).

[88] Intel Corporation. *Intel® Architecture Instruction Set Extensions and Future Features. Programming Reference.* Online. Ref. # 319433-048. Santa Clara, CA, USA, Dec. 2022. URL: `https://cdrdv2.intel.com/v1/dl/getContent/671368` (visited on 16/01/2023).

[89] Apple Inc. *WWDC 2020. Explore the new system architecture of Apple silicon Macs.* Online. Cupertino, CA, USA, June 2020. URL: `https://developer.apple.com/videos/play/wwdc2020/10686` (visited on 20/01/2023).

[90] Linus Torvalds. *Linux kernel source tree. Merge tag 'arm64-upstream' of `git://git.kernel.org/pub/scm/linux/kernel/git/arm64/linux`.* Online. Commit 533b220. June 2020. URL: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=533b220f7be4e461a5222a223d169b42856741ef` (visited on 03/07/2023).

[91] The Clang Team. *Clang 16.0.0 documentation. ShadowCallStack.* Online. Mar. 2023. URL: `https://releases.llvm.org/16.0.0/tools/clang/docs/ShadowCallStack.html` (visited on 03/07/2023).

[92] Kristen Carlson Accardi. *Linux kernel mailing list. [PATCH v3 00/10] Function Granular KASLR.* Online. Message ID <20200623172327.5701-1-kristenlinux.intel.com>. June 2020. URL: `https://lore.kernel.org/lkml/20200623172327.5701-1-kristen@linux.intel.com/` (visited on 29/05/2023).

[93] Apple Security Engineering and Architecture (SEAR). *What if we had the SockPuppet vulnerability in iOS 16?* Online. Apple Inc., May 2023. URL: `https://security.apple.com/blog/what-if-we-had-sockpuppet-in-ios16/` (visited on 01/07/2023).

[94] Jiewen Yao, Vincent J. Zimmer and Jian Wang. *A Tour Beyond BIOS. Security Enhancement to Mitigate Buffer Overflow in UEFI.* Online. Revision 02.0. TianoCore, Mar. 2018. URL: `https://github.com/tianocore-docs/ATBB-Mitigate_Buffer_Overflow_in_UEFI/blob/59e6799291edfb0686e9b20f4bc1251deafa041f/draft/ATBB-Mitigate_Buffer_Overflow_in_UEFI-draft.pdf` (visited on 14/07/2023).

[95] Gerd Hoffmann. *Red Hat Bugzilla. grub2 memory allocation is still broken.* Online. Bug 2149020. Nov. 2022. URL: `https://bugzilla.redhat.com/show_bug.cgi?id=2149020` (visited on 15/07/2023).

[96] Evgeniy Baskov. *Linux kernel mailing list. [PATCH v4 00/26] x86_64: Improvements at compressed kernel stage.* Online. Message ID <cover.1671098103.git.baskovispras.ru>. Dec. 2022. URL: `https://lore.kernel.org/lkml/cover.1671098103.git.baskov@ispras.ru/` (visited on 15/07/2023).

[97] Marvin Häuser. *TianoCore Bugzilla. Enforce W^X design principles.* Online. Bug 3326. Apr. 2021. URL: `https://bugzilla.tianocore.org/show_bug.cgi?id=3326` (visited on 15/07/2023).

[98] Marvin Häuser. *TianoCore Bugzilla. DxeCore: No support for granular PE section permissions.* Online. Bug 3329. Apr. 2021. URL: `https://bugzilla.tianocore.org/show_bug.cgi?id=3329` (visited on 15/07/2023).

[99] Scott Townsend. *BIOS Boot Specification.* Online. Version 1.01. Irvine, CA, USA: Compaq Computer Corporation, Phoenix Technologies Ltd. and Intel Corporation, Jan. 1996. URL: `https://web.archive.org/web/20110715081320/http://www.phoenix.com/resources/specs-bbs101.pdf` (visited on 05/11/2022).

[100] the coreboot project. *coreboot documentation. AMD Family 17h in coreboot.* Online. 4.20. May 2023. URL: `https://doc.coreboot.org/soc/amd/family17h.html` (visited on 15/07/2023).

[101] Apple Inc. *Apple Platform Security.* Online. Cupertino, CA, USA, May 2022. URL: `https://help.apple.com/pdf/security/en_US/apple-platform-security-guide.pdf` (visited on 11/07/2023).

[102] Jiewen Yao and Vincent J. Zimmer. *Understanding the UEFI Secure Boot Chain*. Online. Revision 01.0. TianoCore, June 2019. URL: https://github.com/tianocore-docs/Understanding_UEFI_Secure_Boot_Chain/blob/6921b9415def5260a3170e0dcbe08eadf012b0a9/release-1.00/Understanding_UEFI_Secure_Boot_Chain-release-1.00.pdf (visited on 14/07/2023).

[103] Microsoft Corporation. *Windows Secure Boot Key Creation and Management Guidance*. Online. May 2022. URL: https://learn.microsoft.com/en-us/windows-hardware/manufacture/desktop/windows-secure-boot-key-creation-and-management-guidance?view=windows-11 (visited on 03/12/2022).

[104] Microsoft Corporation et al. *Secure the Windows boot process*. Online. Aug. 2022. URL: https://github.com/MicrosoftDocs/windows-itpro-docs/blob/5e501367a7c1fbc8c6ef14855afd81e50ad298d8/windows/security/information-protection/secure-the-windows-10-boot-process.md (visited on 03/12/2022).

[105] Mikhail Krichanov and Vitaly Cheptsov. 'UEFI virtual machine firmware hardening through snapshots and attack surface reduction'. In: *2021 Ivannikov Ispras Open Conference (ISPRAS)*. Dec. 2021, pp. 30–36. DOI: 10.1109/ISPRAS53967.2021.00010.

[106] TianoCore et al. *EDK II Module Writer's Guide*. Online. DRAFT FOR REVIEW, 12/01/2020 06:45:47. Dec. 2020. URL: https://github.com/tianocore-docs/edk2-ModuleWriteGuide/raw/0a06933f733c4532545ab31120917e221226c03c/draft/edk2-ModuleWriteGuide-draft.pdf (visited on 29/04/2023).

[107] Intel Corporation. *Intel® Firmware Support Package (FSP)*. Online. Commit 3beceb0. June 2023. URL: https://github.com/intel/FSP/tree/3beceb01f90dbdbe7781c7fa43ecc928cd68d2f0.

[108] Nate DeSimone. *edk2-discuss. Re: [edk2-devel] [edk2-discuss] GSoC Proposal*. Online. Message ID 956. Intel Corporation, Apr. 2022. URL: https://edk2.groups.io/g/discuss/message/955 (visited on 06/07/2023).

[109] Andrew Fish. *edk2-discuss. Re: [edk2-devel] [edk2-discuss] GSoC Proposal*. Online. Message ID 956. Apple Inc., Apr. 2022. URL: https://edk2.groups.io/g/discuss/message/956 (visited on 06/07/2023).

[110] Intel Corporation. *X86-S. External Architectural Specification*. Online. Revision 1.0. Document Number: 351407-001. Santa Clara, CA, USA, Apr. 2023. URL: https://cdrdv2.intel.com/v1/dl/getContent/776648 (visited on 25/05/2023).

[111] Agner Fog. *Calling conventions for different C++ compilers and operating systems*. Online. Copenhagen, Denmark: Technical University of Denmark, Aug. 2022. URL: https://www.agner.org/optimize/calling_conventions.pdf (visited on 03/11/2022).

[112] Arm Limited. *Procedure Call Standard for the Arm® Architecture.* Online. Cambridge, England, UK, Oct. 2022. URL: `https://github.com/ARM-software/abi-aa/releases/download/2022Q3/aapcs32.pdf` (visited on 03/12/2022).

[113] Arm Limited. *Procedure Call Standard for the Arm® 64-bit Architecture (AArch64).* Online. Cambridge, England, UK, Oct. 2022. URL: `https://github.com/ARM-software/abi-aa/releases/download/2022Q3/aapcs64.pdf` (visited on 03/12/2022).

[114] Arm Limited. *ARM® Compiler armcc User Guide. Advantages of natural data alignment.* Online. Version 5.06. Cambridge, England, UK. URL: `https://developer.arm.com/documentation/dui0472/m/Compiler-Coding-Practices/Advantages-of-natural-data-alignment` (visited on 03/12/2022).

[115] Microsoft Corporation et al. *pack pragma.* Online. Aug. 2022. URL: `https://github.com/MicrosoftDocs/cpp-docs/blob/883e1a87e5d93b9c45f998d14d6bc01f0fa74dfe/docs/preprocessor/pack.md` (visited on 15/11/2022).

[116] Shedletsky. 'Comment on the Sequential and Indeterminate Behavior of an End-Around-Carry Adder'. In: *IEEE Transactions on Computers* C-26.3 (1977), pp. 271–272. DOI: `10.1109/TC.1977.1674817`.

[117] 'IEEE Standard for Floating-Point Arithmetic'. In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: `10.1109/IEEESTD.2019.8766229`.

[118] ISO. *ISO/IEC 9899:2018 Information technology — Universal coded character set (UCS).* Sixth. Geneva, Switzerland: International Organization for Standardization, Dec. 2020. URL: `https://www.iso.org/standard/76835.html` (visited on 16/07/2023).

[119] Tim Lewis. *edk2-devel. Re: [RFC] Expose HII package list via C variables.* Online. Message ID 79848. Insyde Software, Aug. 2021. URL: `https://edk2.groups.io/g/devel/message/79848` (visited on 08/05/2023).

[120] Microsoft Corporation et al. *Symbol Files.* Online. Jan. 2021. URL: `https://github.com/MicrosoftDocs/win32/blob/59142fb13773208b101d86e0c71886de71e056b6/desktop-src/Debug/symbol-files.md` (visited on 09/05/2023).

[121] Microsoft Corporation et al. *microsoft-pdb. Update README.md.* Online. Apr. 2022. URL: `https://github.com/microsoft/microsoft-pdb/commit/54b4d6370d2faea6aefeaa9fd62724fcc0d47bdd` (visited on 09/05/2023).

[122] Marvin Häuser. *MastersThesis. Supplementary material for the Master's Thesis 'Designing a Secure and Space-Efficient Executable File Format for the Unified Extensible Firmware Interface'.* Online. July 2023. URL: `https://github.com/mhaeuser/MastersThesis/` (visited on 03/07/2023).

[123] Aristotelis Koutsouridis. *Generating a Tiny Test Suite with Greedy Set Cover Minimization*. Online. ForAllSecure, Oct. 2021. URL: `https://forallsecure.com/blog/efficient-corpus-minimization` (visited on 16/07/2023).

[124] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei and Zuoning Chen. 'CollAFL: Path Sensitive Fuzzing'. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, pp. 679–696. DOI: `10.1109/SP.2018.00040`.

[125] Acidanthera. *ocbuild*. Online. Commit 2620958. June 2023. URL: `https://github.com/acidanthera/ocbuild/tree/2620958c4e5b45d1555f846d589580271e8d70e7` (visited on 03/07/2023).

[126] Acidanthera. *OcBinaryData*. Online. Commit 4e7e1b7. June 2023. URL: `https://github.com/acidanthera/OcBinaryData/releases/tag/images-winpe-1.0` (visited on 03/07/2023).

[127] Docker Inc. *Docker*. Online. URL: `https://www.docker.com` (visited on 03/07/2023).

[128] The Linux Foundation ®. *Open Container Initiative*. Online. URL: `https://opencontainers.org` (visited on 03/07/2023).

[129] Inc. GitHub. *GitHub Actions*. Online. URL: `https://github.com/features/actions` (visited on 05/07/2023).

[130] Akihiro Suda et al. *repro-get. Reproducible apt/dnf/apk/pacman, with content-addressing*. Online. Commit 036c30e. May 2023. URL: `https://github.com/reproducible-containers/repro-get/tree/036c30e0c71d71bd8df72b5009934c686db6719b` (visited on 03/07/2023).

[131] Ard Biesheuvel. *edk2-devel. [PATCH v2 0/7] Add PPI to manage PEI phase memory attributes*. Online. Message ID 105649. Google LLC, June 2023. URL: `https://edk2.groups.io/g/devel/message/105649` (visited on 09/05/2023).

[132] Linux Test Project. *LCOV*. Online. Commit 938c8c7. July 2023. URL: `https://github.com/linux-test-project/lcov/tree/938c8c753aa61e87300f8e5f74447066d185829e` (visited on 06/07/2023).