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

Master’s Thesis Presentation by Marvin Häuser

2nd August 2023

Rheinland-Pfälzische Technische Universität
Kaiserslautern
Landau

Outline

1. Introduction to UEFI and Images
2. Image File Format Design
   - Compressed Encoding Techniques
   - Enforcing Constraints via Compressed Encoding
3. Implementations
   - Image File Format Conversion
   - Intermediate Representation
   - Functional and Safety Analyses
4. Evaluation

RPTU

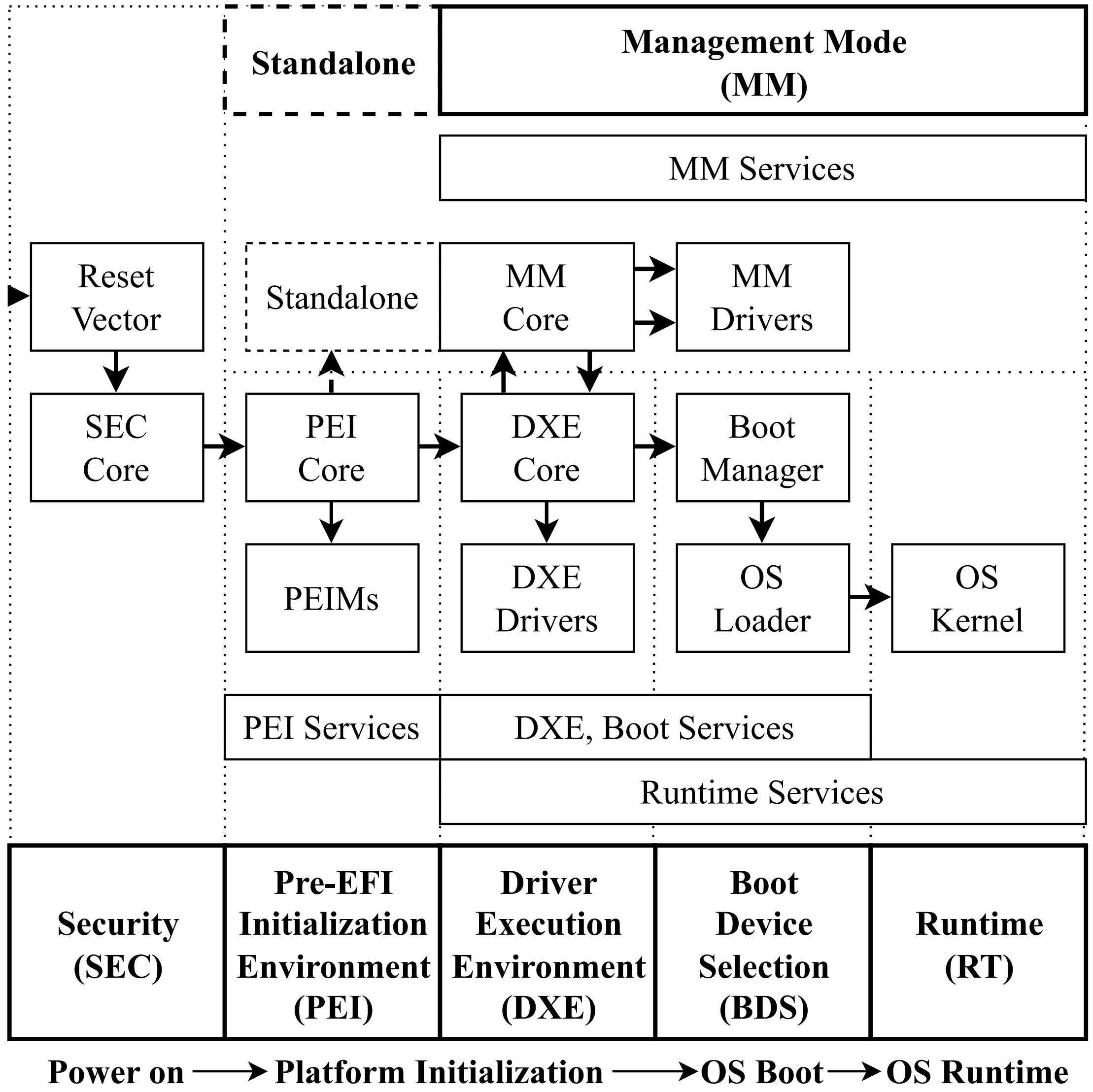
Introduction and Problem Statement

RPTU

UEFI PI and UEFI

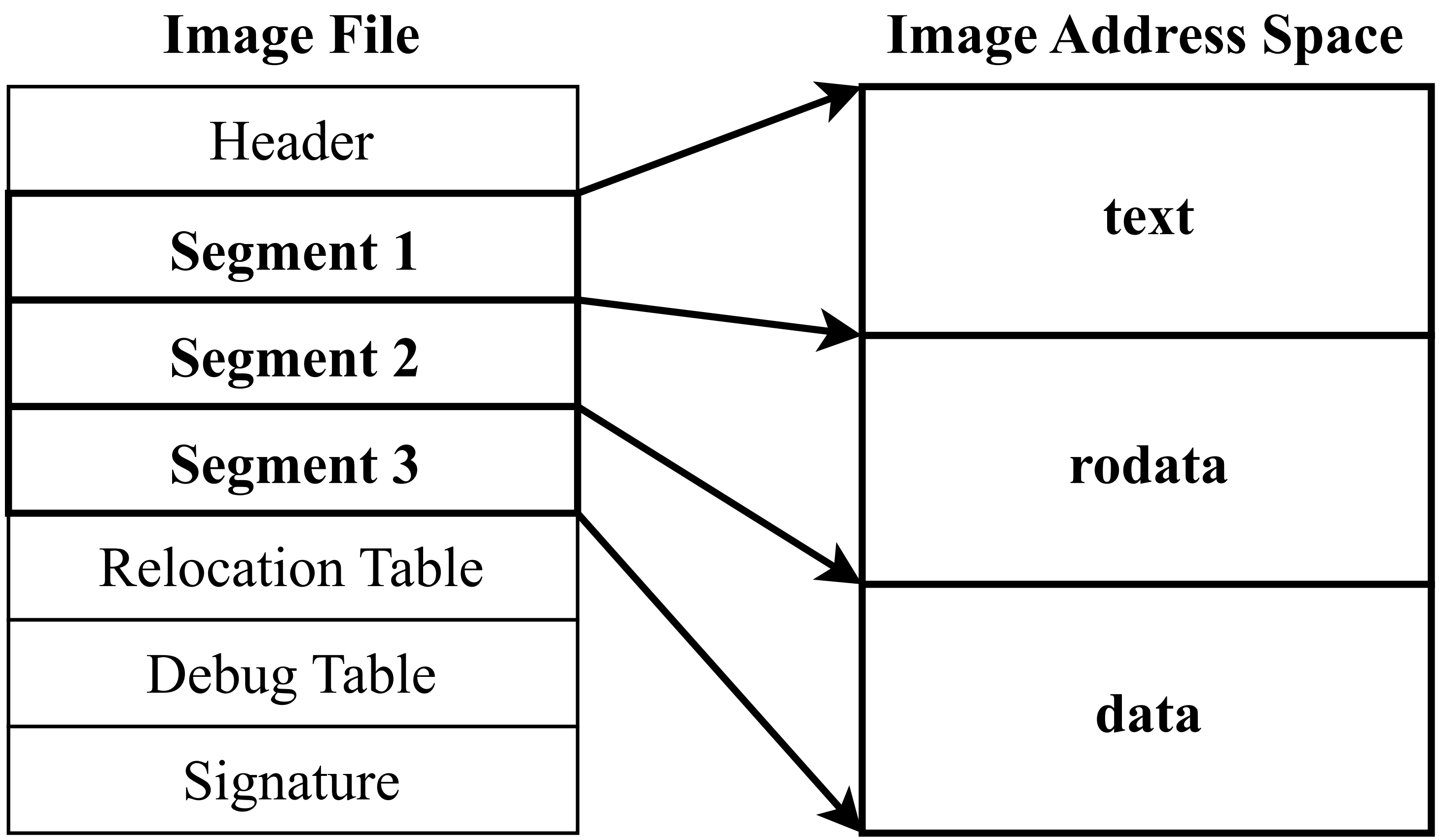
- Unified Extensible Firmware Interface (UEFI) is the de-facto standard boot specification for many private and corporate devices (x86, Qualcomm, …)
- UEFI Platform Initialization (UEFI PI) is the de-facto standard hardware initialization specification for x86 devices
- Both started replacing x86 BIOS at a scale in the early 2010s
- All known widespread implementations are based on TianoCore EDK II
- Support for several security technologies like Secure Boot

UEFI PI and UEFI (Cont'd)



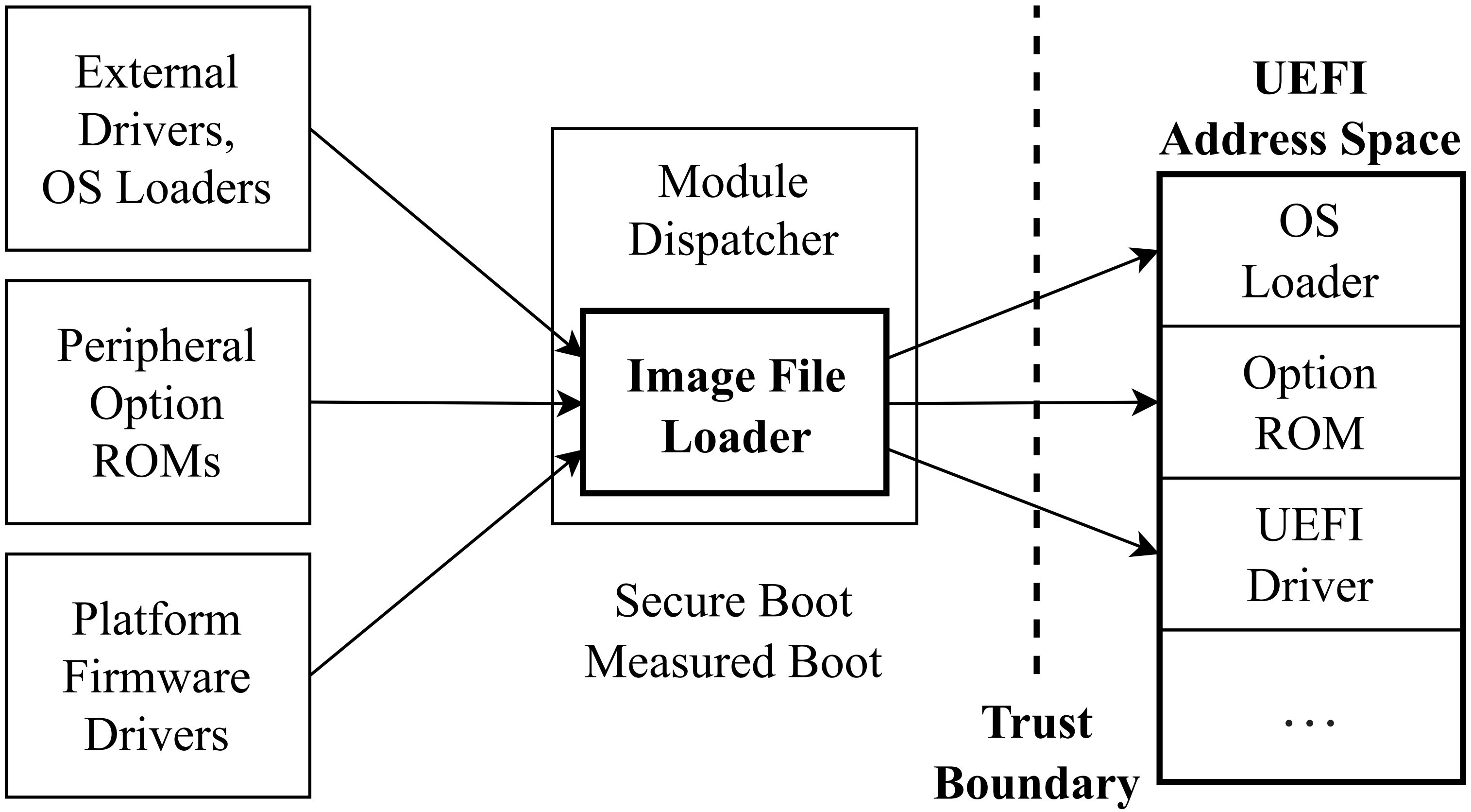
RPTU

Abstraction of UEFI Images

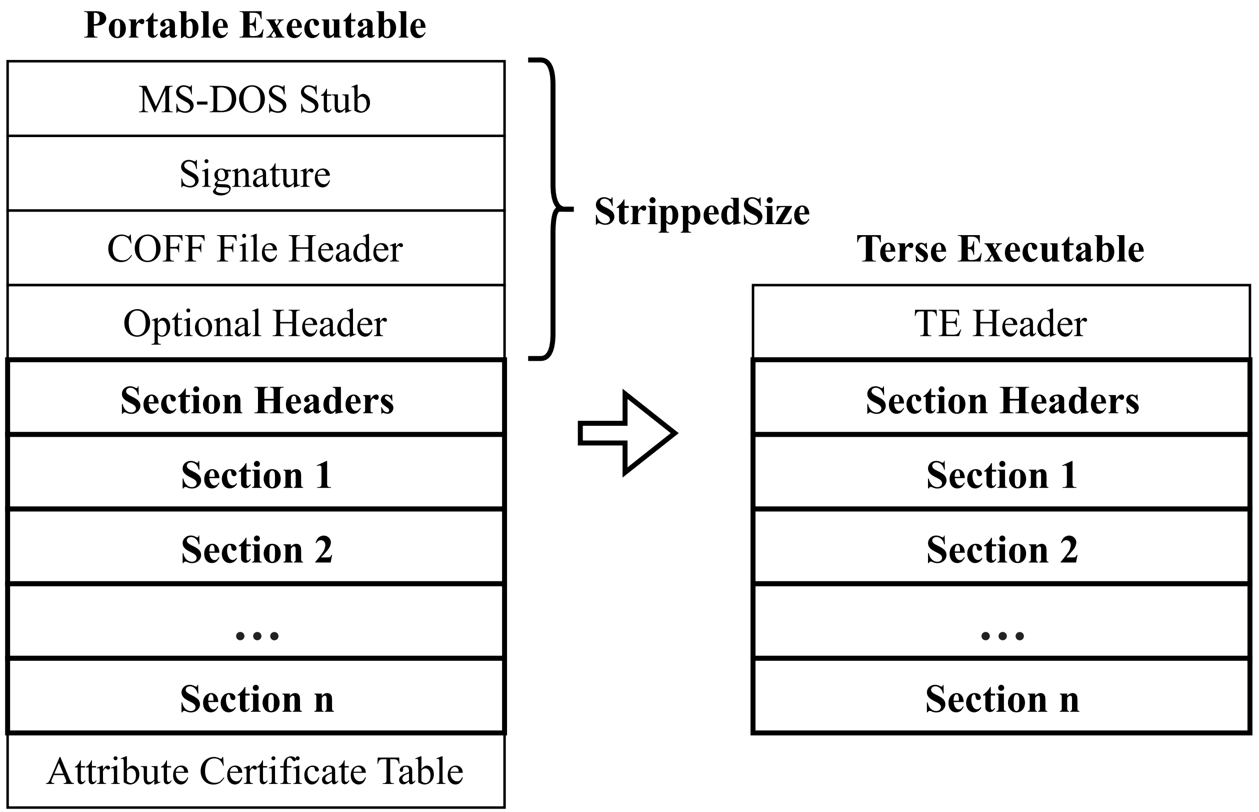


RPTU

UEFI Image File Loader



UEFI PI Terse Executable File Format



Shifted file offsets are not fixed up!

RPTU

Problem Statement

- PE headers are complicated and carry a lot of legacy burden
  - Hard to construct and parse correctly (see real-world bugs)
  - Waste space with unused data (e.g. DOS header)
- UEFI features only few of modern OS security techniques
- Firmware storage generally is very limited → TE format
- Various real-world bugs with the current parsing and generation stacks

→ **Goal**: Design a new stack for firmware internals

- Easy-to-parse and space-efficient UEFI executable file format
- Sophisticated and self-validating image file format conversion

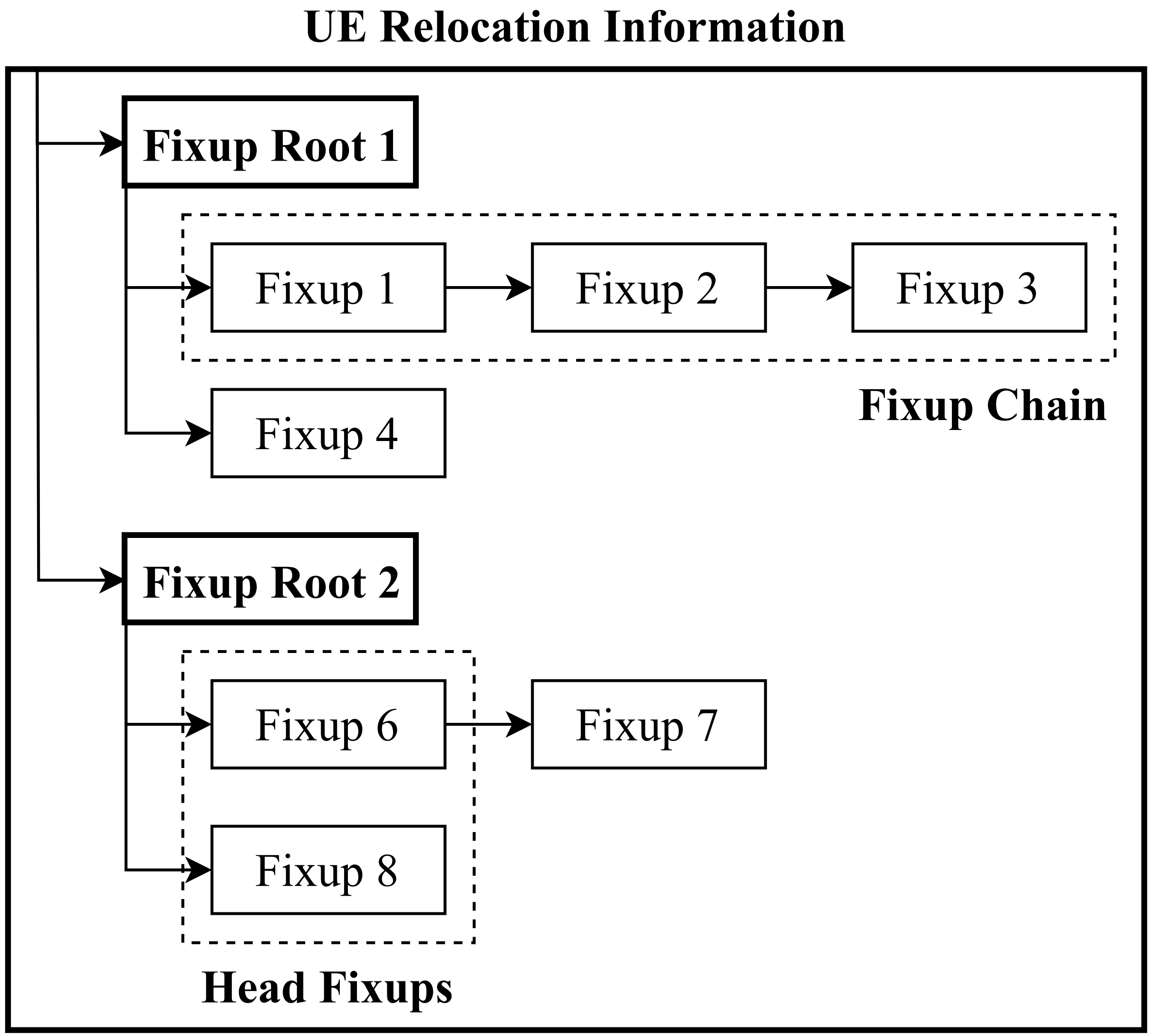
Image File Format Design

RPTU

UEFI Executable File Format

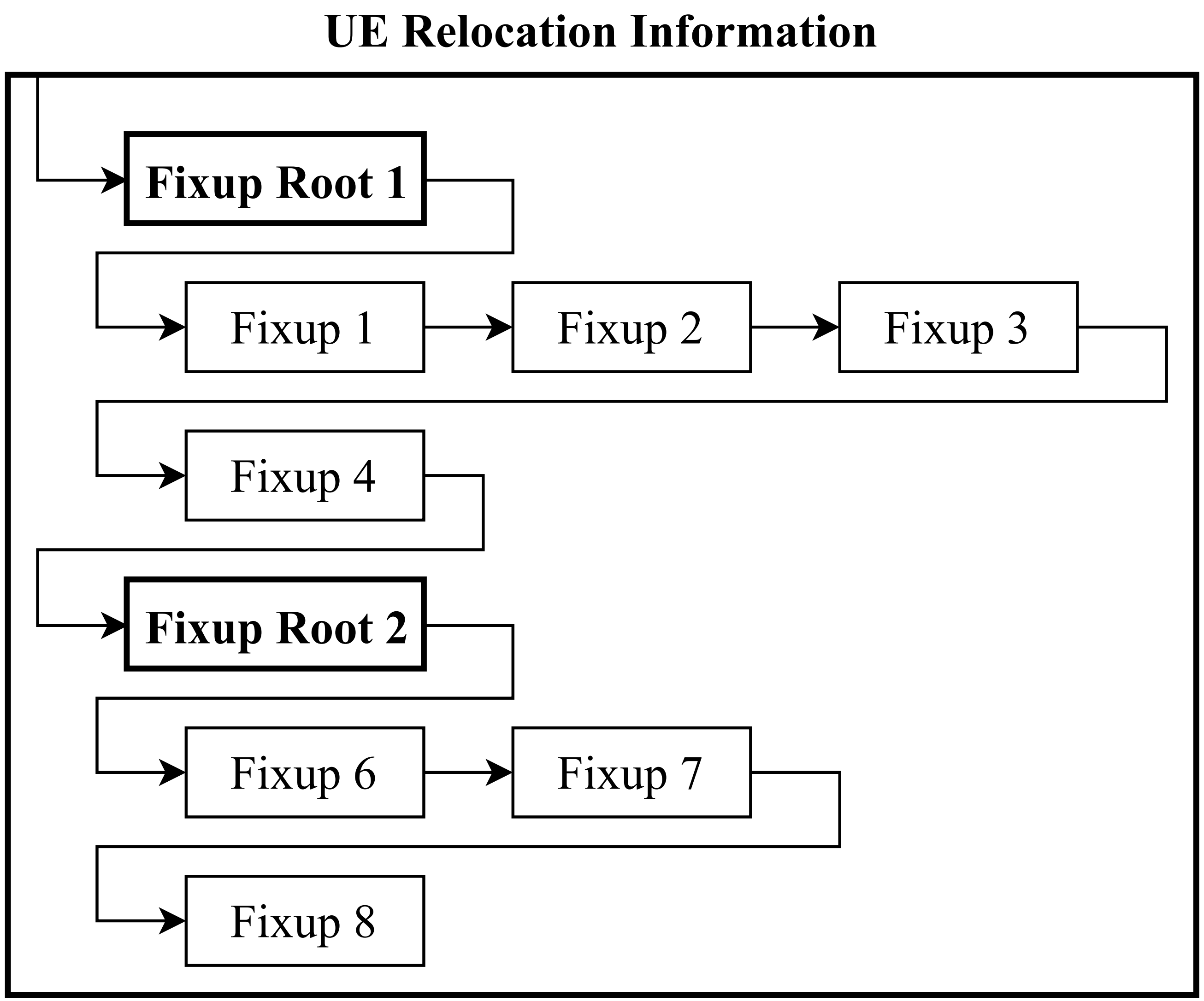
- Limited scope:
    - No support for static or dynamic linking
    - No support for features relying on virtual memory
    - Tailored to UEFI PI and UEFI designs and use cases
- Simple design to enable easy and secure parsing
- Compressed encoding techniques to help with firmware storage constraints
- Unstable design for firmware-internal use

Relocation Fixup Hierarchy



RPTU

Relocation Fixups: Delta Encoding



Relocation Fixups: Delta Encoding (Cont’d)

- Noteworthy space-savings (in firmware storage terms):
    - Only fixup roots use full 32-bit offsets, head fixups use 16-bit offsets
    - Chained fixups store all metadata at the target → no size overhead
- But also, strong guarantees by the encoding:
    - Disjoint targets
    - Ordered ascending
- Can aid formalization efforts
- Particularly slow random access, but there is no UEFI use case

RPTU

Effects of Compressed Encoding

Compressed Encoding

Space Savings

Considered overengineering

Decoding Constraints

Interesting side effect

Enforcing Constraints via Compressed Encoding

- `encode(x)`: Valid Values → Encoded Values
- `decode(x)`: Encoded Values → Decodable Values
- May be well-defined for larger (co-)domains, but output may be nonsense
- If `encode` and `decode` are bijections, Decodable Values = Valid Values
    - Invalid values simply cannot be encoded
    - No runtime checks for validity necessary

Example 1: Data Alignment Requirements

- Data alignment requirements must be a power of two
  - Basic data types: Commonly ‘natural alignment’ (i.e. size = alignment)
  - Image segments: Commonly platform page size (min. 4 KiB for UEFI)
- Idea:
  - For data alignment requirements, encode as $\mathrm{log}_2(\mathrm{x})$
  - For aligned values, encode as $\mathrm{x}\ /\ \mathrm{a}$

RPTU

Example 2: Memory Permissions

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

Memory permissions are typically encoded with a bitmask.

Example 2: Memory Permissions

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

1. Eliminate violations of the W^X rule.

RPTU

Example 2: Memory Permissions (Cont'd)

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

1. Eliminate violations of the W^X rule.
2. **Eliminate unhelpful configurations.**

Example 2: Memory Permissions (Cont’d)

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

We are left with a power of two → encode as enumeration.

Implementations

RPTU

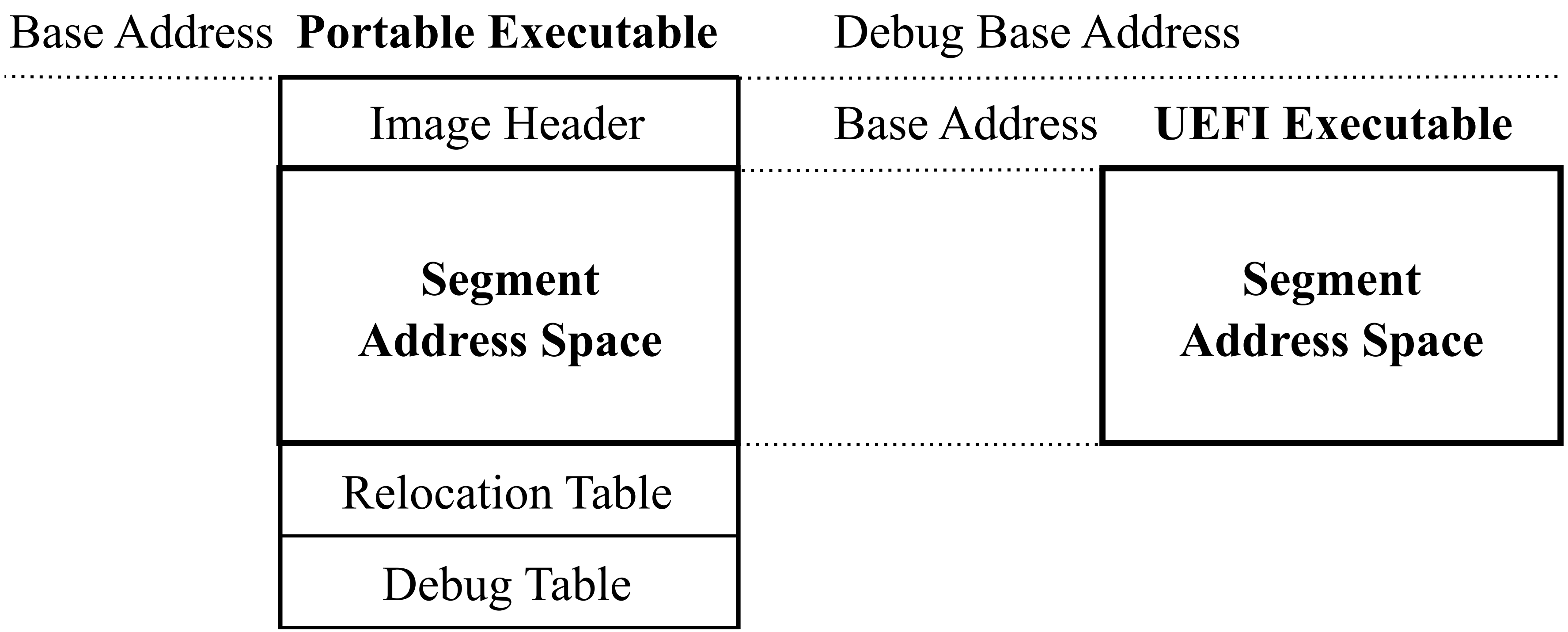
Image File Parsing and Format Conversion

- Existing parsing libraries are tailored to their format
  - Different API design
  - Burden on the caller to support all libraries
- Generate UE files from ELF, Mach-O, and PE files
- Existing conversion solutions translate directly
  - Requires separate translation logic for each input-output pair
  - No output correctness validation

→ **Idea**: Leverage image abstraction for both
  - Generic UEFI image parsing library API following abstraction
  - Conversion via an intermediate representation following abstraction

RPTU

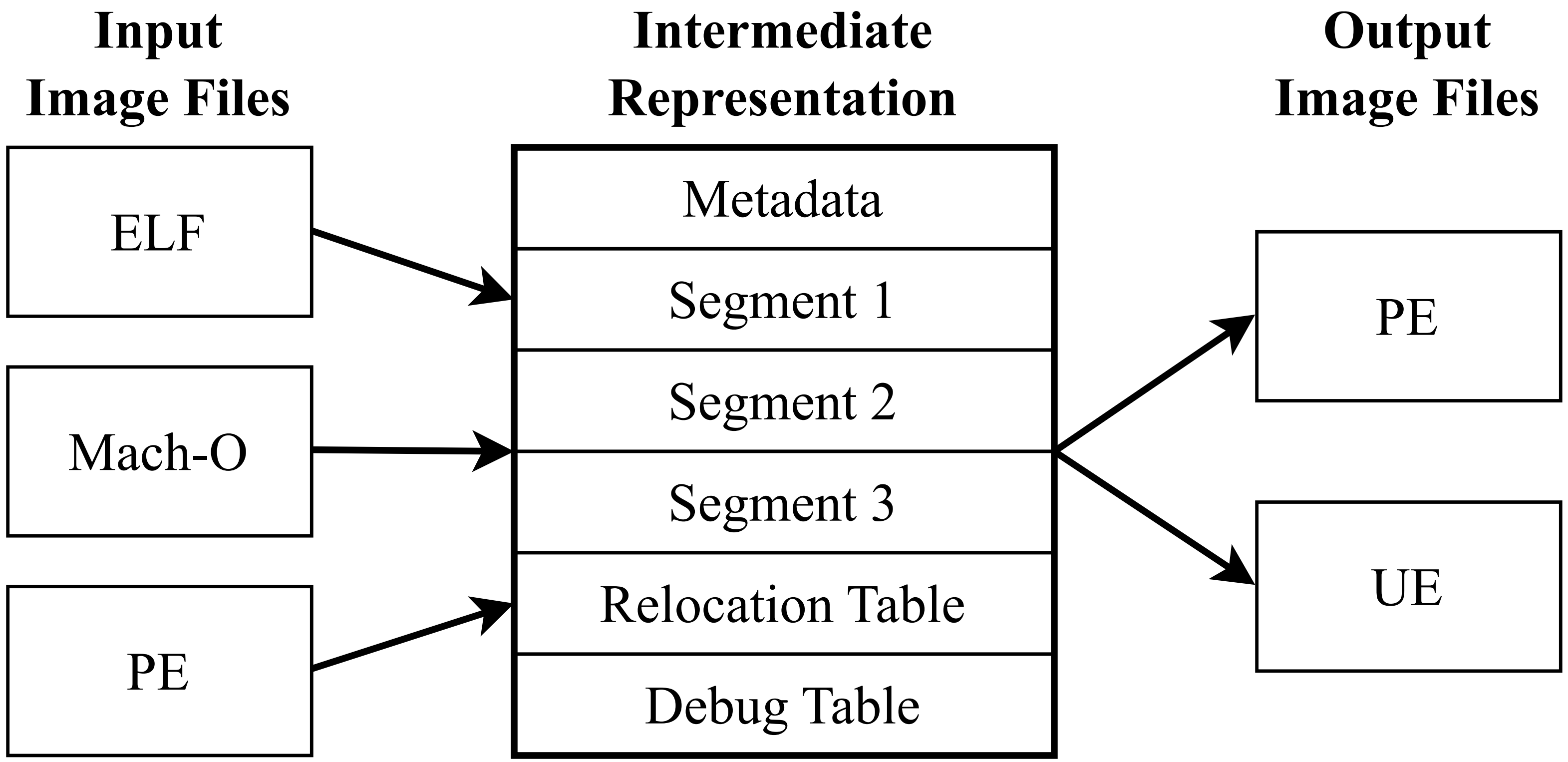
Caveat: PE Implicit File Header Loading



Store the delta in the header.

RPTU

UEFI Image Intermediate Representation

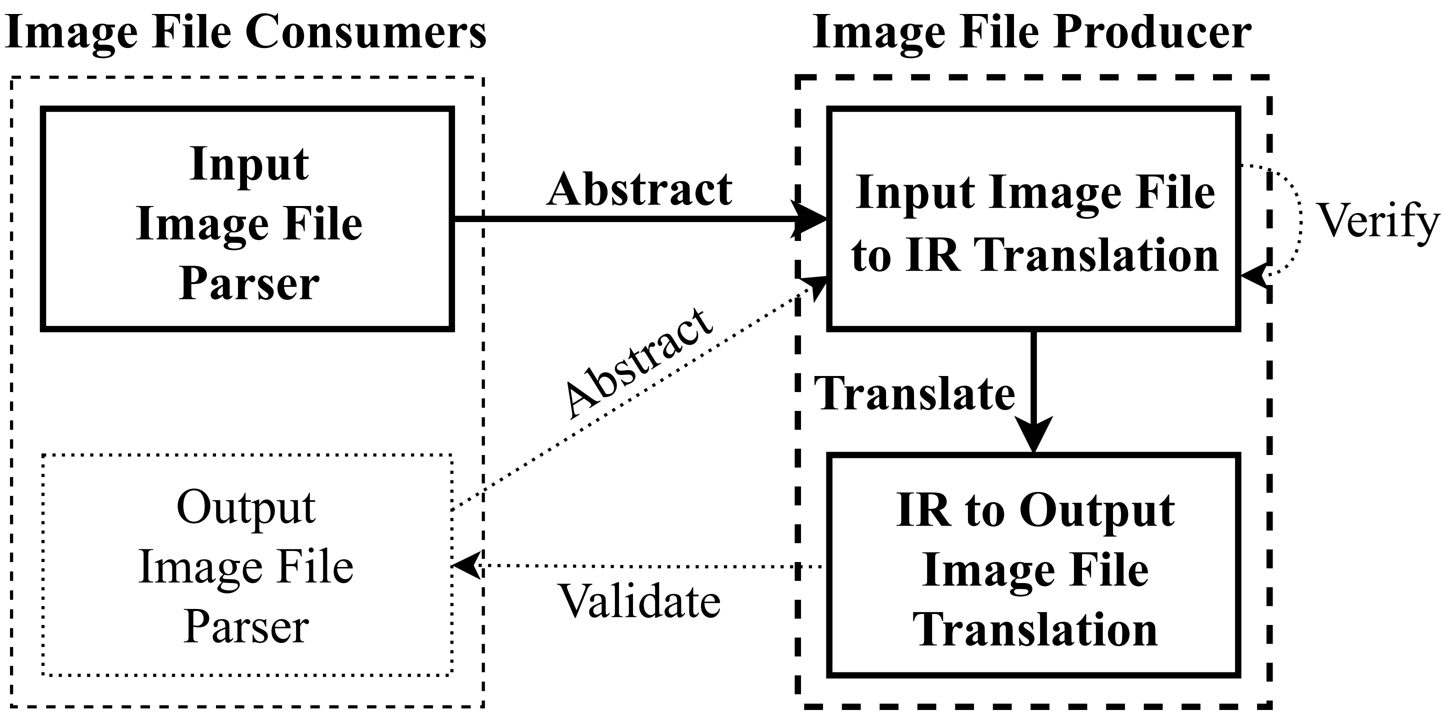


RPTU

UEFI Image Intermediate Representation (Cont’d)

- Based on our UEFI image abstraction
- Semi-canonical definition
  - Most metadata are trivially canonical
  - Segments and reloc fixups are ordered ascending by address
  - Due to format-specific limitations, segment names may be truncated
- Can be used to compare conversion input and output
  - Almost all data are checked for equality
  - Segment names are prefix-matched

UEFI Image Conversion Stack



Functional and Safety Analyses

- Conversion equivalence checking
- Using EDK II parsing libraries to keep requirements in-sync
- Static Analysis using Clang Static Analyzer, GitHub CodeQL, and Coverity
- Fuzz-Testing of the EDK II parser and the conversion using LLVM libFuzzer
    - Functionality testing due to conversion equivalence check
    - Safety testing using LLVM ASan and UBSan
- Automated boot testing using UEFI Shell, Linux, and Windows environments

Evaluation and Conclusion

- All static analysis reports on main components were resolved
- Fuzz-testing slowed down corpus evolution after two weeks
  → Best-effort functional and safety validation
- 9.7 % avg. space saving per module, up to 78 KiB total for X64 OVMF
  → Evolutionary space efficiency improvements
- Visibly smaller parsing library compared to PE
- All artifacts were publicly released and all tests were run by GitHub CI

→ The task was solved, but only a first step towards simplifying booting

Thank You!

RPTU