SoK: All You Ever Wanted to Know About Bootloader Security But Were Afraid to Ask

Connor Glosner
Purdue University

Aravind Machiry
Purdue University

***Abstract*—Bootloaders are present in every device, from IoT and edge devices to datacenter servers, making them critical to system security. Modern platforms establish hardware root of trust via vendor-specific mechanisms such as CPU microcode and authenticated code modules before bootloader execution. This SoK focuses on the software boot chain that follows that hardware-rooted integrity handoff. Prior work has focused on subsets of bootloaders, often using inconsistent terminology, leaving gaps in understanding their structure and interactions.**

We analyze 43 bootloaders and categorize them into three types based on their architecture and role in the boot process. We define attack surfaces for each type using our open-source dataset, BOOTBENCH**, which includes 3,658 vulnerabilities and associated commits. Leveraging** BOOTBENCH**, we evaluate existing vulnerability detection techniques and highlight open problems, and examine limitations in defensive techniques for hardening bootloaders.**

1. Introduction

Bootloaders are foundational to modern computing systems, bridging the gap between a system's hardware and an Operating System (OS), hypervisor (*e.g.,* Microsoft Hyper-V [94], VMware ESXi [19]), or embedded application (*e.g.,* Zephyr [92]). They execute after hardware-established root of trust and extend the chain of trust into software [8], yet vulnerabilities can grant attackers persistent system control [49], [87], [143]. Despite their critical role, bootloaders remain highly vulnerable [55], [128], [156]. In 2024 alone, 204 CVEs were reported, some leading to remote code execution in privileged environments [49], [65], [101], [102].

Existing works [55], [71], [121], [128], [156], [170] attempt bootloader vulnerability detection, but focus on narrow components or attack surfaces. For instance, the recent RSFUZZER [170] focuses only on SMI handlers and cannot find PIXIEFAIL [49] and LOGOFAIL [143], while FUZZUER [55] focuses on boot-time interfaces and cannot find LOGOFAIL. More broadly, there is little understanding of bootloader attack surfaces, and most tools are tailored to specific implementations (*e.g.,* EDK-II [148], GRUB [57], or aboot [4]). General-purpose analysis techniques [54], [67], [125], [154], [171] exist, but are hard to configure and ignore bootloader-specific challenges as demonstrated in our analysis (§ 4).

Bootloaders further complicate security because they lack common OS abstractions and run before features such as virtual memory are initialized, limiting the applicability of standard exploit mitigation and software hardening techniques [2], [14], [30], [44], [97], [132]. While efforts like E3C [20] attempt to automatically add spatial safety to EDK-II through Checked C [39], their generality to other bootloaders is unknown. Overall, *there is no systematic view of bootloader types, attack surfaces, or the effectiveness of existing security techniques*.

In this paper, we present the first systematization of bootloader security. We begin with background on booting and bootloaders, collecting a dataset of 43 bootloaders and classifying them into three types by initialization and handoff. Next, we identify 6 attack surfaces and examine their relevance across bootloader types. We then categorize vulnerability detection techniques and evaluate their effectiveness, showing that only 3 of 25 tools cover 2 of 6 attack surfaces. Since no prior dataset exists, we built BOOTBENCH, comprising 3,658 vulnerabilities, which reveals that tools are often specialized to particular bootloaders, detect few vulnerability classes, and generalize poorly. Finally, we study prevention and hardening techniques, finding that exploit mitigations are hard to apply and proactive security remains rare. While some approaches exist [22], [31], [88], [152], they are limited in scope or are disabled [9].

In summary, the following are our contributions:

- **Bug Dataset (§ 4.2):** BOOTBENCH, the first open dataset of 3,658 bootloader vulnerabilities, commits, and tools.
- **Evaluation:** Qualitative and quantitative evaluation (using BOOTBENCH) of existing techniques (§ 4) to detect limitations and open-problems.
- **Study:** First comprehensive study of modern bootloaders (§ 2) and their attack surfaces (§ 3).
- **Systematization (§ 4, § 5):** A systematic analysis of vulnerability detection techniques (static and dynamic) and defences (exploit mitigation, security hardening, and vulnerability prevention), identifying characteristics, challenges, and open problems.

2. Booting and Bootloaders

Bootloaders have been defined differently across domains [121], [135], [140], [156]. For example, Redini *et al.,* [121] focus on smartphone bootloaders, while Wang *et al.,* [156] examine OS-specific bootloaders.

At a high level, *booting is the process of starting a system and preparing it for target software*, and the boot-

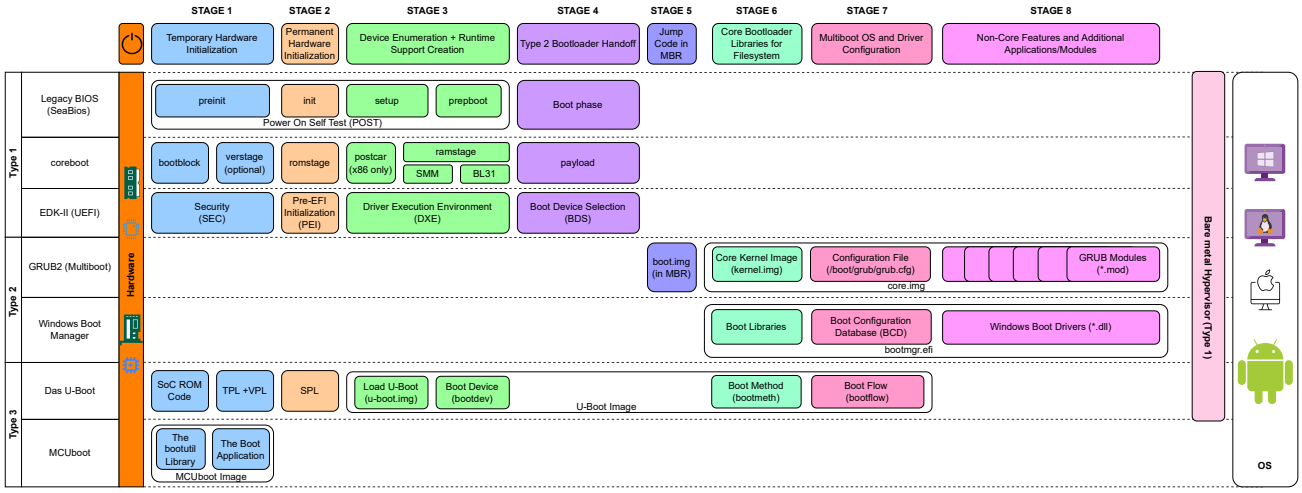

Figure 1: Bootloader Boot-Process Comparison.

loader is the software that performs this task. Based on their starting point and target software, bootloaders can be classified into three types, Figure 1:

- **Firmware Bootloader (Type 1)**: Boots from hardware, providing a hardware-agnostic interface for subsequent software (*e.g.,* UEFI, BIOS). It may load another bootloader or standalone applications.
- **OS Bootloader (Type 2)**: Boots from a system-initialized state (*e.g.,* BIOS) and prepares for an OS (*e.g.,* GRUB2). In staged booting, type 1 hands off to type 2.
- **Monolithic Bootloader (Type 3)**: Combines type 1 and type 2, booting directly from hardware into an OS. Common in embedded systems, these are hardware-specific.

In contrast to traditional bootloaders, **universal kernel images (UKIs)** [133] encapsulate the kernel, `initramfs`, and command-line data into a single, signed artifact that can be loaded directly by firmware or OS bootloaders, simplifying the boot chain and improving portability across platforms. Functionally, UKIs resemble *type 2 bootloaders* since they rely on firmware (type 1) to initialize hardware and then prepare the operating system. However, because UKIs integrate all boot components (*i.e.,* the kernel, `initramfs`, and configuration) into a single image rather than maintaining them as separate artifacts, we treat them as a streamlined subclass of type 2 bootloaders for the purposes of our discussion.

2.1. Split v/s Monolithic Booting

Two approaches exist to boot an OS or hypervisor, Figure 1:

- **Split (Staged) Booting:** Type 1 + Type 2. Provides modularity and hardware abstraction, easing OS initialization, but increases firmware size and startup time. Common in desktops, servers, and smartphones.
- **Monolithic Booting:** Type 3. Smaller and faster, but tightly coupled to hardware, limiting flexibility. Common in resource-constrained devices such as IoT and MCUs.

2.1.1. Booting Stages and Diversity

The entire booting process can be divided into 8 stages, as seen at the top of Figure 1, each of which sets up the necessary hardware and software for the following stages. Not all stages appear in every bootloader. For example, Stage 4 (bootloader handoff) is absent in type 3 bootloaders, as they are monolithic, while Stages 5–8 are absent in Type 1, since it remains OS-agnostic. There is considerable diversity in the implementation of these different stages. Stage implementations may vary and include sub-components, enabling customization for different hardware and software configurations. For instance, coreboot uses multiple components to implement stage 3, whereas in EDK-II (UEFI), it is encapsulated in a single component (*i.e.,*DXE).

2.2. Bootloaders Dataset

We systematically identified bootloaders across domains by surveying prior research [55], [128], [136], [156], public documentation cited by those works [32], [47], [58], and curated sources such as the Open-Source Firmware Foundation [48]. This yielded 43 bootloaders, both open and closed source. Each bootloader was manually analyzed to identify its features, supported architectures, and type classification. We summarize our bootloaders in our extended report [1]. The full dataset of bootloaders is organized at https://github.com/BreakingBoot/oss-bootloaders.

For the remainder of this paper, we focus on 7 out of the 43 bootloaders, seen in Table 1 and in Figure 1. These bootloaders were selected based on their relevance in recent work [55], [128], [136], [156], the number of known vulnerabilities [126], and their distinct design characteristics [26], [91]. We acknowledge that this selection may introduce

Table 1: Well-Known Bootloaders. Refer to Appendix A for the notations meanings. We annotate CI/CD integration with icons representing the type of automated analysis: unit testing framework ( ), CodeQL static analysis ( ), Coverity static analysis ( ), Cppcheck static analysis ( ), OSS-Fuzz continuous fuzzing ( ), LLVM libFuzzer integration ( ), OpenSSF security scorecard checks ( ). A full extended table of bootloaders is provided in our extended report [1] and our dataset repository.

Name	Supported OSs	Supported Architectures	O.S.S. [Stars + Forks]	CI/CD Integration	Attack Surfaces
Type 1: Firmware Bootloaders					
coreboot [24]		RISC-V, x86, ARM, PPC	✔[3k]		has_1, has_2, sas_1 sas_2, sas_3, sas_4
EDK-II (UEFI) [148]		x86, ARM, RISC-V, MIPS, LoongArch	✔[8.1k]		has_1, has_2, sas_1 sas_2, sas_3, sas_4
SeaBios [25]		x86	✔[587]	✖	has_1, has_2, sas_1 sas_2, sas_3, sas_4
Type 2: OS Bootloaders					
GRUB [57]		x86, ARM, PPC, MIPS, SPARC, RISC-V, LoongArch	✔[404]		has_1, has_2, sas_1 sas_2, sas_4
Windows Boot Manager [157]		x86, ARM	✖	-	has_1, has_2, sas_1 sas_2, sas_4
Type 3: Monolithic Bootloaders					
MCUboot [90]	RTOS	32bit MCU	✔[2.4k]		has_1, sas_2 sas_3
Das U-Boot [153]	RTOS	ARM, x86, ARC, MIPS, PPC, RISC-V, Xtensa, sh, microblaze, m68k, nios2	✔[8.6k]		has_1, sas_1, sas_2 sas_3, sas_4

bias, as other bootloaders also feature unique designs [13], [80]. However, we believe this subset has sufficient diversity to illustrate the core concepts of bootloader design. Our analysis aimed to answer two questions: *how users interact with each bootloader* (§ 2.3) and *how control is transferred during the boot sequence* (§ 2.4). We provide complete case studies on each of the 7 bootloaders in our extended report [1].

2.3. Bootloader Communication Mechanisms

Communication across different boot phases is critical for transferring configuration state, coordinating execution, and enabling later stages or payloads to interact with the underlying firmware. Across the boot ecosystem, we identify three broad communication mechanisms: *system tables*, *non-volatile storage*, and *configuration files*. Some bootloaders also employ interrupts or well-defined memory regions for direct low-level control, particularly in legacy environments.

Structured Handoff via System Tables and Interrupts Type 1 bootloaders such as UEFI [47] and SeaBIOS [25] implement explicit interfaces for communication between boot stages. UEFI defines system tables that expose *Boot Services* and *Runtime Services*, allowing later phases to query device paths, protocol handles, and persistent configuration stored as NVRAM variables (*e.g.,* boot order or Secure Boot options) [47], [69]. SeaBIOS, following legacy BIOS conventions, instead relies on interrupts such as `INT 0x19` to locate and load subsequent stages [60], [64]. coreboot [24] itself provides no user-facing communication interface. Instead, its payload (*e.g.,* UEFI or SeaBIOS) defines the mechanisms through which the state is passed or retrieved.

Dynamic Configuration via External Files Type 2 bootloaders shift communication into higher-level configuration data structures or files. Windows Boot Manager [157] stores boot paths, recovery modes, and drivers in the Boot Configuration Data (BCD), which can be modified at runtime (*e.g.,* pressing `F11` triggers recovery mode through BCD flags) [158]. Similarly, GRUB2 [57] defines its menu entries, kernel arguments, and chainloading options in `grub.cfg`. These text-based configurations enable flexible runtime communication, allowing parameters such as the Linux `root` device or alternate payloads (*e.g.,* Windows Boot Manager) to be specified without rebuilding the bootloader.

Static Communication and Minimal Runtime Interfaces In contrast, type 3 bootloaders integrate initialization, hardware setup, and OS handoff within a single image, seen with Das U-Boot [153] and MCUboot [90] in Figure 1, leaving few standardized channels for runtime interaction. MCUboot primarily supports firmware update communication through alternate flash slots, without exposing system tables or runtime interfaces [91]. Conversely, U-Boot provides a minimal runtime shell that permits limited reconfiguration of the `bootflow` or kernel arguments, but substantial changes require reflashing [32]. Overall, these tightly coupled bootloaders rely on static configurations and direct flash management rather than dynamic communication during execution.

2.4. Bootloader Handoff

The handoff phase marks a critical transition where control passes from one execution context to another, typically from a type 1 to a type 2 bootloader or directly to the kernel. It must preserve essential system state (*e.g.,* memory maps, hardware tables, configuration structures) while

releasing firmware-managed resources to let the next stage assume control. The complexity of this process varies across designs, reflecting architectural goals and legacy support.

Structured and Layered Handoff Type 1 bootloaders (*e.g.,* UEFI [47], SeaBIOS [25], coreboot [24]) employ formal transition mechanisms. UEFI's Boot Device Selection (BDS) phase, for instance, iterates through the `BootOrder` variable stored in the NVRAM to locate a boot application, execute it, and invoke `ExitBootServices()` to release firmware resources and finalize control transfer [88]. SeaBIOS follows legacy BIOS conventions via `INT 0x19` to load and jump to the MBR or VBR [88]. coreboot hands control to a payload (*e.g.,* Depthcharge [59] or a UEFI stub) that builds system tables and initializes runtime services [139]. In essence, UEFI enforces a clean resource-managed handoff, BIOS chains control through sector loaders, and coreboot delegates transitions to modular payloads.

Configurable OS Handoff Type 2 bootloaders, such as Windows Boot Manager [157] and GRUB2 [57], emphasize flexible chainloading and configuration. `bootmgr.efi` transfers control to `winload.efi` with UEFI tables, BCD entries, and preloaded drivers. GRUB2 loads the target OS kernel (*e.g.,* Linux's `vmlinux` and `initrd`) or chainloads into Windows Boot Manager. This configurability enables multi-OS booting and dynamic reconfiguration without rebuilding the bootloader.

Direct and Minimal Handoff Type 3 bootloaders couple initialization and OS launch in a single flow, passing minimal runtime data. MCUboot [90] verifies firmware and directly jumps to the OS (*e.g.,* Zephyr [92]) [91]. U-Boot [153] offers a richer interface—providing OS image, arguments, and FDT, but performs a mostly static transfer [32]. These tightly integrated bootloaders prioritize simplicity and determinism over extensibility.

3. Bootloader Attack Surfaces

Bootloaders form a critical part of the Trusted Computing Base (TCB), extending a hardware-established root of trust into software. On modern platforms, vendor-specific mechanisms (*e.g.,* Intel Boot Guard ACMs [70], AMD PSP [27], ARM TrustZone [5]) verify early boot integrity before bootloader code executes. Vulnerabilities in this layer can enable persistent compromises (bootkits [88]), which survive OS reinstallation and even disk replacement, often requiring complete hardware reformatting to remove.

As described in § 2, bootloaders interact with both hardware and software components, exposing multiple attack surfaces at boot time (*i.e.,* non-runtime) and after boot (*i.e.,* runtime). We classify these broadly into hardware and software attack surfaces as summarized in Table 2.

3.1. Hardware Attack Surfaces

Here, we assume that the attacker has physical access to the system running the bootloader. Depending on the level of access, these can be further classified into:

- **Invasive Hardware Access (has_1)** — Attackers with invasive (*e.g.,* dismantling the system) physical access can perform processor-level or memory attacks such as *clock glitching [134] or cold boot [82]*. We scope this surface to *non-modifying* invasive attacks (*i.e.,* fault injection, glitching, and side-channel probing) rather than hardware modification (*e.g.,* reflashing a custom firmware image), which would allow the attacker to replace the entire boot image, defeating the purpose of analyzing the existing attack surface. For instance, NAND flash glitching can compromise U-Boot [151], while MoonBounce [76] implants malicious code in the SPI flash of EDK-II systems. Also, FinFisher [99] is a surveillance-based malware that corrupts the host machine through the MBR on the HDD that targets Windows Boot Manager, GRUB, and Android.
- **External Hardware Access (has_2)** — Non-invasive access (*e.g.,* plugging in peripherals) can also serve as attack vectors. For instance, ThunderStrike2 [93] implants malware in Thunderbolt accessories that are loaded at boot as OptionROMs in EDK-II [148], creating persistent backdoors. Additionally, CVE-2020-25647 [119] is triggered during USB device initialization and can lead to arbitrary code execution and Secure Boot bypass in GRUB [57] (type 2 bootloader).

3.2. Software Attack Surfaces

Here, attacker interacts with the bootloader through software interfaces. Depending on the type of interface, these can be further classified into:

- **Remote Access (sas_1)** — Network boot features expose bootloaders to remote attacks. For instance, PixieFAIL [49] and The Real Shim Shady (CVE-2023-40547 [120]) exploit protocol parsing bugs in EDK-II's [148] and shim [145] PXE boot, respectively, to achieve remote code execution. Similarly, CVE-2018-18439 [98], is a remotely exploitable buffer overflow in the U-Boot [153] (type 3 bootloader) that leads to remote code execution if the TFTP server is controlled.
- **Persistent Data Source Access (sas_2)** — Attackers with write access to persistent storage can tamper with configuration files or boot partitions. For instance, BootHole [34] (CVE-2020-10713 [118]) abuses a vulnerability in GRUB's [57] `grub.cfg` parser, while LogoFAIL [143] exploits EDK-II's [148] logo parsing by changing the OS logo. Similarly, CVE-2019-13104 [100] in U-Boot [153] allows crafted ext4 filesystems to overwrite stack data during boot.
- **Post-boot Features Access (sas_3)** – As mentioned in § 2.3, some bootloaders expose runtime handlers (*e.g.,* SMI callbacks [47]) that remain active post-boot and serve as potential attack surfaces. For instance, TrickBoot [35] leverages these in EDK-II and SeaBIOS [126] to extract protected SPI flash information. For type 3 bootloaders, it can be as simple as rewriting part of the flash memory from the devices OS allowing for complete control of the infected device as seen with UbootKit [166] in U-Boot.

Table 2: Attack Surfaces Classification by Bootloader Type.

Bootloader Types	Attack Surfaces					
	Hardware		Software			
	Invasive Hardware (has_1)	External Hardware (has_2)	Remote Access (sas_1)	Persistent Data Source (sas_2)	Post-boot Features (sas_3)	Boot Time Feature (sas_4)
Type 1	✔ (MoonBounce [76])	✔ (ThunderStrike2 [93])	✔ (PixieFAIL [49])	✔ (LogoFAIL [143])	✔ (TrickBoot [35])	✔ (CVE-2024-7756 [83])
Type 2	✔ (FinFisher [99])	✔ (CVE-2020-25647 [119])	✔ (The Real Shim Shady [36]) (CVE-2023-40547 [120])	✔ (BootHole [34]) (CVE-2020-10713 [118])	✖	✔ (CVE-2024-49504 [141])
Type 3	✔ (Glitching U-Boot [151])	✖	⚠ (CVE-2018-18439 [98])	✔ (CVE-2019-13104 [100])	⚠ (UbootKit [166])	⚠ (CVE-2023-48426 [33])

- **Boot time Features Access** (sas_4) — Interactive boot-time interfaces (*e.g.,* UEFI or GRUB shells [88]) also form another class of attack surfaces. For instance, CVE-2024-7756 [83] (EDK-II [148]), CVE-2024-49504 [141] (GRUB [57]), and CVE-2023-48426 [33] (U-Boot [153]) all exploit vulnerabilities through boot time interfaces.

Table 2 summarizes our classification and prevalence of attack surfaces across different types of bootloaders.

Finding 1

Not all attack surfaces are present across bootloaders. type 1 and type 2 bootloaders exhibit consistent surfaces, while type 3 bootloaders show greater variability.

4. Vulnerability Detection

In this section, we evaluate vulnerability detection techniques for bootloaders and their integration into the bootloaders presented in Table 1. An adoption study can be seen in § 4.4.3 and § 4.5.3 for static and dynamic analysis techniques, respectively.

4.1. Techniques Selection

To evaluate existing vulnerability detection techniques, we examined papers from the top four security (IEEE S&P, ACM CCS, USENIX Security, ISOC NDSS) and software engineering conferences (ICSE, FSE, ASE, OOPSLA) over the past decade [172] and extracted referenced tools and frameworks. This resulted in a total of 25 tools, as summarized in Table 3. We excluded ML-based detectors since the only prior work (Microsoft [71]) lacked an accompanying evaluation framework.

Preliminary classification: We classify detection techniques in Table 3 as either static or dynamic. Additionally, we have collected generic analysis techniques [54], [67], [125], [131], [154], both static and dynamic, that either have been adapted in other works [67], [125] or can be adapted [54], [131], [154] to work with bootloaders. Symbolic execution is treated as static since it analyzes paths without execution. For each technique, we record the targeted attack surface, vulnerability type, bootloader type tested, and desired features for an ideal technique (*e.g.,* Automated (A)). Some approaches combine static and dynamic elements [55], [165], [170], but we classify by the primary detection methodology. For example, UEFUZZER [165] uses static type inference for input generation but relies on dynamic execution to expose bugs. In total, we identify 10 static and 15 dynamic techniques.

Most existing vulnerability detection tools focus on software attack vectors, specifically, boot-time (sas_4) and post-boot (sas_3) surfaces, while largely neglecting hardware (has_1, has_2) and remote/persistent data (sas_1, sas_2) vectors. This is emphasized by the examples we presented for these attack surfaces in Table 2, where has_1, has_2, sas_1, and sas_2 can lead to arbitrary code execution.

Finding 2

Prior works primarily target boot-time and post-boot interfaces, leaving other hardware and software attack surfaces underexplored.

Observation 1

Our analysis also revealed no comprehensive bootloader vulnerability dataset. Existing works either evaluate only on upstream bootloaders [121], [167], [170] or on limited (<10) known vulnerabilities [55], [156].

To address this gap, we created the BOOTBENCH dataset.

4.2. BOOTBENCH

We aim to collect real-world vulnerabilities to enable quantitative evaluation of existing techniques. We compiled a real-world vulnerability dataset by mining the CVE database [28] and identified a total of 1,157 bootloader CVEs. Previous work [84] shows that security bugs could be fixed without CVEs. To handle this, we mined the commit histories of open-source bootloaders from our dataset and used keyword-based filtering (derived from MITRE's CWE list [96]) to identify a total of 3,658 non-CVE security bugs, covering 43 bootloaders.

The most common vulnerability types are privilege escalation, memory overflows, and information disclosure. Legacy BIOS (367), EDK-II (357), GRUB (74), and U-Boot (55) dominate the dataset. We acknowledge possible omissions (missed CVEs, unrelated third-party commits) and discuss limitations in Appendix 8.

4.3. Evaluation Methodologies

For both static (§ 4.4) and dynamic (§ 4.5) analyses, we followed two consistent methodologies: one for selecting

the analysis tools, and another for choosing the subset of vulnerabilities from BOOTBENCH used for evaluation.

- **Tool Selection:** Based off of the tools in Table 3, we selected those that were open-source, well-documented, and runnable without specialized hardware. Several tools were excluded due to execution failures (*e.g.,* arbiter [154]) or missing documentation (*e.g.,* FuzzBoot [156]). Each selected tool was executed on representative bootloader source code and binaries (when supported), following official documentation with minimal configuration changes, which we discuss these changes in Appendix B. We recorded all outputs (*i.e.,* crashes, logs, and vulnerability reports) and manually validated each reported issue against our injected ground truth to distinguish true positives (TPs) from false positives (FPs). The final tools used in our evaluation are listed in Table 5.
- **Ground Truth Data Selection:** We constructed the ground truth dataset from known bootloader vulnerabilities included in BOOTBENCH. Not all bootloaders or vulnerabilities were evaluated. Several analysis tools require source-level access, compiler instrumentation, or specific binary formats, which are not uniformly available across all bootloaders. Consequently, we selected a representative subset of seven bootloaders that span all three bootloader classes and for which the necessary analysis workflows were feasible. Similarly, we did not include every reported vulnerability. Many disclosed issues correspond to obsolete commits or code paths that have since been refactored or removed in the latest version. To ensure consistency and reproducibility, we selected a single commit per bootloader that contained the largest number of simultaneously present, verifiable vulnerabilities. This enabled direct pre- and post-patch comparison using CVE-linked fixes to assess each tool's accuracy. Table 7 in Appendix B summarizes the evaluated bootloaders, their commit hashes, and the number of known vulnerabilities at each chosen revision.

4.4. Static Analysis

As shown in Table 3, we collected a total of 10 static vulnerability detection techniques and tools.

Out of which, 7 (top of the table) are bootloader-specific techniques (BSP) (*i.e.,* these are custom-designed to handle BSP aspects) as indicated by the **BT** column. Furthermore, these focus on specific classes of vulnerability seen by the **TBT** column. Most of these tools are interface aware (**IA** column), *i.e.,* they are aware of interfaces exposed by the bootloader and model them appropriately. This is expected as they focus on specific attack surfaces (**Attack Surfaces** column) and mainly model the corresponding interfaces. For instance, SPENDER [169] focuses on the post-boot interfaces attack surface (*i.e., sas*$_3$) and appropriately handles the corresponding interfaces (*i.e.,* SMI handlers). This is not the case with FwHunt [11], which uses BSP patterns to detect vulnerabilities. It is interesting to see that *most of the BSP tools (except for STASE [128] and SymUEFI [10]) directly work on bootloader binaries (**BO** column), thereby side-stepping the issues of configurability and build setup*. From a usage perspective, *tools are not automated and require non-trivial effort to set up, except for STASE [128] and FwHunt [11].* Furthermore, a few of these are not open-source (**O.S.S** column), and we did not receive any response from the authors when contacted for the tool access.

Listing 1 Interface Invocation Challenges. ➊$_1$: Structures vary across attack surfaces. ➋$_2$: `VOID*` cast to different structures. ➌$_3$: Even within protocols, required fields differ and may be state-dependent.

```
➊typedef struct{UINT32 KeyId;UINT8 Data[32];}CREATE_KEY_COMM;
➊typedef struct{ UINT32 KeyId; } DELETE_KEY_COMM;

...CreateKeyHandler(...,VOID *CommBuffer,UINTN *CommBufferSize){
➋  CREATE_KEY_COMM *Req = (CREATE_KEY_COMM *)CommBuffer;
  // ... use Req->KeyId / Req->Data ...
  return EFI_SUCCESS;
}

...DeleteKeyHandler(...,VOID *CommBuffer,UINTN *CommBufferSize){
➋  DELETE_KEY_COMM *Req = (DELETE_KEY_COMM *)CommBuffer;
  // ... use Req->KeyId ...
  return EFI_SUCCESS;
}

EFI_STATUS EFIAPI OpenFileSystemExample(EFI_HANDLE ImageHandle){
  EFI_SIMPLE_FILE_SYSTEM_PROTOCOL *Fs; EFI_FILE_PROTOCOL *Root;
  gBS->HandleProtocol(..., &gEfiSimpleFileSys..., (VOID **)&Fs);
➌  Fs->OpenVolume(Fs, &Root);
  // ... work with Root ...
  return EFI_SUCCESS;
}

EFI_STATUS EFIAPI RngExample(EFI_HANDLE ImageHandle) {
  RNG_PROTOCOL *Rng; UINT8 Buf[16]; UINTN Size = sizeof(Buf);
  gBS->HandleProtocol(..., &gEfiRngProtocolGuid, (VOID **)&Rng);
➌  Rng->GetRNG(Rng, NULL, Size, Buf);
  // ... use random bytes ...
  return EFI_SUCCESS;
}
```

There are two general-purpose techniques (*i.e.,* CodeQL [54] and arbiter [154]), but these do not model the BSP aspects (*e.g.,* interfaces and taint sources) and require BSP configurations and customizations to be effective. However, unlike BSP tools, these are fairly easy to set up and run.

4.4.1. Fundamental Aspects

We describe the fundamental requirements of static analysis techniques and unique challenges in bootloaders. We also analyze how existing techniques try to tackle each of these challenges and identify open problems. We mainly focus on the capabilities needed for vulnerability detection [123], specifically: dataflow tracking, configurability, and robust language support. Table 4 summarizes the static analysis capabilities of existing tools for different attack surfaces by bootloader type.

Dataflow Tracking: One of the fundamental capabilities in static vulnerability detection is the ability to track the flow of interesting data (*i.e.,* tainted data) to different program points, commonly referred to as *Taint Tracking* [81]. For example, the GRUB2 BootHole vulnerability (CVE-2020-10713) [34] stems from attacker-controlled configuration data flowing into a flex-generated parser. Since GRUB2 redefines `YY_FATAL_ERROR()` to return unexpectedly, an unchecked input reaches `yy_flex_strncpy()`, which copies it into a heap buffer, causing overflow and potential code execution. Detecting such flaws requires taint tracking that is inter-procedural [46], alias-aware [77], and capable of modeling custom taint sources and sinks [164]. Achieving

Table 3: Feature comparison of bootloader analysis tools. Target Bug Type (TBT): Memory Corruption (MC), Storage State (SS), Temporal Vulnerabilities (TV), Input Validation (IV), Improper Memory Isolation (IMI), CPU Exceptions (CE), Privilege Escalation (PE), Malware Detection (MD), Invalid Configuration (IC), Pattern Matching (PM). Features: Bootloader Type (BT), Bootloader Agnostic (BA), Binary Only (BO), Hardware Independent (HWI), Interface Aware (IA), Automated (A), Open Source (O.S.S). Partial support is shown as ⚠, and `N/A` marks generic tools needing custom setup. See § A for symbol details.

Tool	BT	Attack Surfaces	TBT	BO	HWI	IA	A	O.S.S.
Bootloader-Specific Tools								
Static Analysis Tools								
STASE [128]	1	sas_1,sas_2,sas_3,sas_4	MC, IMI	✗	✓	✓	✓	✓
SPENDER [169]	1	sas_3	IMI	✓	✓	✓	⚠	✗
SymUEFI [10]	1	sas_3	IMI	✗	✓	✓	⚠	✗
Excite [41]	1	sas_3	CE	✓	✗	✓	⚠	✗
FwHunt [11]	1	sas_1,sas_3,sas_4	PM	✓	✓	✗	✓	✓
BootStomp [121]	2	has_2,sas_2,sas_4	MC, SS	✓	✓	✓	✗	✓
EMBA [40]	1,3	sas_1,sas_2,sas_3,sas_4	MC,PE,IC	✓	✓	✓	⚠	✓
Dynamic Analysis Tools								
FuzzUEr [55]	1	has_2,sas_1,sas_3,sas_4	MC,CE	✗	✓	✓	✓	✓
UEFUZZER [165]	1	sas_1,sas_2,sas_3,sas_4	MC	⚠	✓	✓	✓	✗
SimFuzzer [167]	1	has_1,has_2,sas_2,sas_3,sas_4	CE	✗	✓	✗	✗	✗
RSFuzzer [170]	1	sas_3	MC, IV	✓	✓	✓	⚠	✗
UEFIscanner [95]	1	sas_2,sas_3	MD	✓	✗	✗	✓	✗
HBFA [122]	1	has_2,sas_3,sas_4	MC	✗	✓	✓	✗	✓
UEFIFuzzer [124]	1	sas_4	CE	✗	⚠	✓	✗	✓
efi_fuzz [112]	1	sas_2	MC	✓	✓	✗	⚠	✓
BootFuzz [137]	1	sas_3	CE	✓	✓	✗	✗	✓
abootool [62]	2	sas_4	CE	✓	✗	✗	✗	✓
Emmutaler [51]	2	has_2,sas_3,sas_4	CE,[MC⚠]	✓	✓	✗	✗	✓
FuzzBoot [156]	2,[3 ⚠]	has_2,sas_3,sas_4	MC	✗	⚠	✓	✗	✓
ASPFuzz [53]	3	has_1,has_2	CE,MC	✓	✓	✗	✗	✓
General-Purpose Tools								
arbiter [154]	N/A	N/A	N/A	✓	✓	✗	✓	✓
CodeQL [54]	N/A	N/A	N/A	✗	✓	✗	✓	✓
Angr [131]	N/A	N/A	N/A	✓	✓	✗	✗	✓
kAFL [125]	N/A	N/A	N/A	✗	⚠	✗	✗	✓
TSFFS [67]	N/A	N/A	N/A	✗	✓	✗	✗	✓

this in bootloaders is particularly hard due to their pointer-heavy code, stage-dependent control flow, and reliance on persistent state [63]. We highlight four technical challenges in this space.

Static Analysis Technical Challenge 1 (SATC1): *Points-to analysis* [109], [138] – Memory management in bootloaders is done through direct physical address ranges and does not use well-known memory management APIs, such as `malloc`/`free`, which is used by many points-to analysis techniques. Furthermore, to minimize their footprint, bootloader code frequently employs both data and function pointers (*e.g.,* EDK-II uses both, as shown in Listing 1). Resolving the precise targets of such pointers is difficult since they may depend on dynamic configuration [55], [128].

As shown in Table 4, existing works tackle points-to analysis through heuristics, but coverage across attack surfaces remains limited. STASE [128] applies field- and structure-sensitive analysis, but only at a function level. FuzzUEr [55] uses flow-insensitive points-to analysis, but their aliasing model is based on source-level expressions (*i.e.,* two pointers are aliases if they are syntactically the same). SPENDER [169] uses EDK-II-specific heuristics (*i.e.,* `CommBuffer` for SMI handlers) to handle points-to analysis.

Table 4: Static analysis capabilities for different attack surfaces by bootloader type (1, 2, 3). Automated techniques are marked ⚙; ✖ means no existing technique. Tool names can be cross-referenced in Table 3. Parenthetical counts denote known vulnerabilities per surface.

	Necessary Capabilities	Techniques	HW Surfaces		SW Surfaces			
			has_1(2)	has_2(4)	sas_1(12)	sas_2(16)	sas_3(10)	sas_4(45)
Dataflow	Points-to (SATC1)	Heuristic / Field-Sensitive	✖	2:⚙ [121]	1:⚙ [55], [128]	1:⚙ [128] 2:⚙ [121]	1:⚙ [55], [128], [169]	1:⚙ [55], [128] 2:⚙ [121]
	Taint Src/Sink (SATC2)	Annotation / Rule-Based	✖	2:⚙ [121]	1:⚙ [55], [128]	1:⚙ [128], [165] 2:⚙ [121]	1:⚙ [41], [55], [128], [169]	1:⚙ [11], [55], [128] 2:⚙ [121]
	Interrupt Flow (SATC3)	Callback / Event Modeling	✖	✖	✖	1:⚙ [165]	1:⚙ [55], [128], [165], [169] 2:⚙ [121]	1:⚙ [55], [165] 2:⚙ [121]
	Cross-stage (SATC4)	State / Memory Tracking	✖	✖	✖	1:⚙ [165] 2:⚙ [121]	1:⚙ [41], [55], [165], [169]	1:⚙ [55], [165] 2:⚙ [121]
	Configurability	Build / Config Handling	✖	2:⚙ [121]	1:⚙ [55], [128]	1:⚙ [128] 2:⚙ [121]	1:⚙ [55], [128], [169]	1:⚙ [11], [55], [128] 2:⚙ [121] 3:⚙ [40]
	Language Support	Source (C/C++)	1:⚙ [10], [54], [55], [128] 2,3: ✖					
		Binary	1:⚙ [11], [40], [41], [169] 2:⚙ [121] 3:⚙ [40]					
		Cross-Language (C+ASM)	✖					

Finding 3

No existing static vulnerability detection tool tries to perform systematic points-to analysis (an important capability for taint tracking), which could lead to unsound results, as seen in Table 5.

Static Analysis Technical Challenge 2 (SATC2): *Taint source and sink identification* [175] – Unlike traditional software, bootloaders lack well-documented taint sources and sinks, and their use of MMIO, assembly stubs [17], and *cross-domain* data propagation complicates identification. For instance, values written to NVRAM in a type 1 bootloader stage (*e.g.,* EDK-II DXE) may be read later by another (*e.g.,* GRUB), requiring the analysis to model inter-bootloader data dependencies.

As shown in Table 4, tools address this in various ways: FuzzUEr [55] uses argument annotations to model sources and sinks, UEFUZZER [165] recovers structures from binary analysis, STASE [128] applies vulnerability-specific rules, SPENDER [169] uses protocol-based function pointers. Exite [41] monitors SMRAM accesses, while BootStomp [121] models NVRAM flows but is limited to Android bootloaders.

Open-Problem 1 (Technical)

There exists no technique to automatically identify taint sources and sinks in bootloaders. The recent advancements in LLMs [78] could enable potential solutions.

Static Analysis Technical Challenge 3 (SATC3): *Interrupt-driven control flow* [16] – Unlike traditional software, bootloader execution is not strictly sequential. Hardware interrupts, firmware-generated exceptions, or asynchronous event introduce non-linear flows invisible in static call graphs [55], [128]. This unpredictability can result in incomplete coverage of execution paths, under-approximation of reachable code, and missed detection of race conditions between initialization routines and event handlers [174].

As shown in Table 4, existing BSP tools handle limited cases such as protocol-based callbacks but do not fully model hardware interrupts or asynchronous callbacks (*e.g.,* FuzzUEr [55], UEFUZZER [165], SPENDER [169], and STASE [128]) BootStomp [121] modeled callbacks and persistent memory accesses with symbolic execution. However, it relied heavily on Android framework semantics, including the predictable structure of boot components and system services, which makes them ill-suited for general bootloader analysis, where control flow can vary across platforms and firmware implementations.

Finding 4

Asynchronous and interrupt-driven control flow is prevalent in bootloaders. Existing techniques handle this in a limited manner, mostly based on specific patterns.

Static Analysis Technical Challenge 4 (SATC4): *Cross-stage state persistence* – Bootloaders operate across multiple execution stages (§ 2) and may pass complex states via system tables, NVRAM, or reserved memory regions. Taint tracking should be able to track such a persistent state across different bootloader stages. Without modeling this persistence, vulnerabilities tied to stale or maliciously modified states (*e.g.,* corrupted memory maps or altered boot parameters) remain undetected. For example, in GRUB if coreboot was the underlying type 1 bootloader, all of the system tables are read in from coreboot by reading register `r0` seen in `grub/grub-core/kern/arm/coreboot/cbtable.c` and could potentially contain tainted data.

As shown in Table 4, existing techniques only partially capture persistence, and typically within a single bootloader or bootloader stage. Some works like FuzzUEr [55] and UEFUZZER [165] explore intra-stage dependencies to recover state transitions within individual stages using structure- or protocol-based heuristics. Other techniques model persistence through a fixed pattern. For instance, SPENDER [169] and Exite [41] handle the state persistence only in SMM or SMRAM. BootStomp [121] extends the idea to Android and models NVRAM as both a persistent store and a taint source. No tool fully models state persistence and taint propagation across multiple stages. While SATC2 concerns

identifying taint entry and exit points, SATC4 extends this to state surviving stage transitions (*e.g.,* via NVRAM or system tables). In both cases, static analysis must reason about inter-stage continuity through transient data flows or long-lived state.

Observation 2

Stages in bootloader exchange state and data through various mechanisms (*e.g.,* specific memory regions, persistent storage, etc). Tracking such data is important for identifying vulnerabilities that span multiple stages or bootloaders.

Configurability: Bootloaders are large and highly configurable, demanding techniques that can scale to thousands of files, multiple architectures, and conditionally compiled options. For instance, EDK-II supports 44 bus drivers across 55 different boards and configurations (*e.g.,* QEMU without network support). Consequently, static analysis techniques that hook onto the build setup, such as CodeQL [54], FuzzUEr [55], STASE [128], may not be able to analyze files that are not part of the compilation. As shown in Table 4, existing techniques use only a specific configuration of the bootloader. However, naively considering all configurations is tedious. Therefore, we need a way to identify a configuration that covers the majority of the codebase. This problem is similar to determining the maximal configuration in the Linux Kernel [3], [104], where several techniques have been explored [38], [50]. However, there exists no such technique for bootloaders.

Observation 3

Bootloaders contain various complex build configurations, and no technique exists to handle this or identify the maximal configuration enabling high static analysis coverage.

Robust programming language support: As shown in Table 4, bootloaders make extensive use of assembly, low-level constructs, and compiler idioms, all of which static analysis must handle for comprehensive coverage.

Static Analysis Engineering Challenge 1 (SAEC1): *Cross language analysis* - Bootloaders frequently mix C with assembly [148] and, in some cases, custom domain specific languages (DSLs) [57] to optimize performance across architectures and board features.

Observation 4

Cross-language support, specifically between assembly and C/C++, is need for effective vulnerability detection in bootloaders. However, none of the existing source-based techniques has such a capability.

4.4.2. Tools Evaluation

Based on the tool selection methodology outlined in § 4.3, we were left with 3 out of the 10 static analysis tools. We evaluated each of these tools on our dataset and analyzed the warnings as mentioned in § 4.3.

Results Tool performance varied considerably, as shown in Table 5. CodeQL and FwHunt were both straightforward to configure, with FwHunt achieving zero false positives due to its extensive pattern filtering, while CodeQL produced only false positives, requiring significant manual triage. For TF-A [6], several injected bugs were detected by Coverity [12] in CI/CD workflows (seen by the patch notes in the commit message), but CodeQL did not detect them. STASE, a hybrid approach combining static analysis with symbolic execution, avoided false positives but was constrained by its limited set of predefined rules, detecting only 5 of the 16 injected vulnerabilities (31%).

Table 5: Bug Finding Ability of Tools Across Bootloaders (✔: Ran, ✖: Failed, (TP/FP): True Positives / False Positives)

Tool	Type 1			Type 2		Type 3	
	EDK-2 (16)	coreboot (8)	SeaBios (3)	GRUB (29)	shim (11)	U-Boot (12)	TF-A (11)
Static Analysis Tools							
CodeQL	✔(0/23)	✔(0/0)	✔(0/5)	✔(0/17)	✔(0/0)	✔(0/56)	✔(0/0)
FwHunt	✔(0/0)	✖	✖	✖	✖	✖	✖
STASE	✔(5/0)	✖	✖	✖	✖	✖	✖
Dynamic Analysis Tools							
FuzzUEr	✔(1/0)	✖	✖	✖	✖	✖	✖
HBFA	✔(0/0)	✖	✖	✖	✖	✖	✖
efi_fuzz	✔(0/0)	✖	✖	✖	✖	✖	✖
BootFuzz	✖	✖	✔(0/0)	✖	✖	✖	✖

Among the evaluated tools, STASE achieved the highest precision and lowest false positive rate, outperforming both CodeQL and FwHunt on EDK-II. FwHunt's portability issues limited its usefulness beyond a single platform, and CodeQL's broad false positive output highlights the need for firmware-specific query refinement. Overall, STASE provided the most reliable results despite its limited rule set, yielding the lowest false negative rate.

Finding 5

Existing static analysis tools fail to detect injected BSP vulnerabilities due to strict heuristics. Detected vulnerabilities are limited to simple memory-related issues; none handle the challenges of SATC3 and SATC4 effectively.

4.4.3. Adoption in Bootloaders

Following prior work showing SAST tools are commonly integrated via CI [130], we manually inspected the CI configurations of open-source bootloaders.

Results. None of the bootloaders use bootloader-specific static techniques. Only 20.1% (9) integrate general-purpose SAST tools, and configurations are typically left at defaults, which Shen *et al.,* [130] show to be largely ineffective without bootloader-specific tailoring.

4.5. Dynamic Analysis

These techniques analyze bootloaders at runtime to detect vulnerabilities. As shown in Table 3, there are 13 dynamic analysis techniques with different capabilities, targeting different types of bootloaders. The majority, 13 (87%), of these are BSP tools. Furthermore, similar to static analysis tools, most of the techniques are for type 1 bootloaders, specifically EDK-II. It is interesting to see that many of the tools work directly on binary firmware (as indicated

by the **BO** column). However, these techniques are specific to an attack surface. For instance, RSFuzzer [170] works directly on EDK-II firmware, but only focuses on sas_3 attack surface (*i.e.,* SMI Handlers). Many of the tools require manual configuration (as indicated by the **A** column). For instance, HBFA [122] requires harnesses to be written for each interface. General purpose tools such as kAFL [125] and TSFFS [67], also require considerable modification to handle bootloader input sources. There are a few automated techniques but not all of them are open-source (*e.g.,* UEFUZZER [165] and RSFuzzer [170]). We did not receive any responses from the authors when we inquired.

4.5.1. Fundamental Aspects

At a fundamental level, dynamic analysis techniques [85] require four necessary capabilities (*i.e.,* target execution, interface invocation, input generation, and invalid state (vulnerability) detection).

Target Execution. The ability to execute the target is the foundation of any dynamic analysis technique. Unlike user-space programs that rely on standard libraries (*e.g.,* `libc`) and run easily in emulators such as QEMU [116], bootloaders are hardware-dependent and are often triggered by interrupts, making reliable execution in virtualized environments challenging [163].

Although bootloaders usually include emulator-specific support, it is limited. For example, barebox, a type 3 bootloader, supports 20 different SPI drivers across all configurations, but only 5 are supported in QEMU, reducing coverage [7]. The diversity of bootloaders further complicates this, since execution strategies must account for differences in architectures (*e.g.,* x86, ARM, RISC-V), board-specific initialization, and firmware interfaces. For instance, initializing permanent memory on an ARM `mach-rockchip` board involves only 4 steps, while the same process on a Freescale `imx8mm_evk` board requires 7 steps due to a more complex setup and unique memory layout.

As shown in Table 6, most tools rely on off-the-shelf simulators, though four [112], [122], [165], [170] only partially emulate the bootloader. On-device execution is rare [62], [95], and *automated rehosting*, which would support COTS bootloaders without vendor-specific support, is entirely absent.

Observation 5

Most techniques use emulators as execution environments, which have limited support for bootloader.

Open-Problem 2 (Engineering)

We need an automated rehosting technique that can handle bootloaders.

Input Generation. Dynamic analysis requires generating effective inputs for target interfaces. Therefore, two things are required: identifying the expected input type or structure, *argument type identification*, and producing values consistent with that type, *argument value generation*.

For example, Listing 1 shows an interface that accepts a `VOID*` pointer and length, which is cast into different concrete structures (*e.g.,* `CREATE_KEY_COMM` or `DELETE_KEY_COMM`) depending on the invoked handler (➔$_2$). Bootloaders expose diverse attack surfaces, so expected input structures vary across interfaces. In EDK-II (type 1 bootloader), SMI handlers (sas_3) typically accept communication buffers cast into command structures, while boot-time service protocols (sas_4) take typed arguments (➔$_3$ vs. ➔$_2$ in Listing 1). Even within a single attack surface, input structures differ across different interfaces. As shown in Table 6, manual grammar specification dominates type identification [51], [53], [122], [156], [167], but it is labor-intensive. For example, HBFA [122] requires custom stubs and harnesses per CVE, limiting coverage of other vulnerabilities. Automated type inference [55], [165], [170] shows promise but remains limited to specific bootloader types and attack surfaces.

For value generation, most works rely on fixed or random values [53], [122], [156], [167], which fail for state-dependent inputs (*e.g.,* a `DELETE_KEY_COMM` requiring a valid previously created `KeyId`). Inference-based approaches [55], [170] address this but remain confined to a few surfaces. Only 30% of techniques consider type 2 or type 3 bootloaders [51], [53], [62], [156], and none automate input generation for them.

Observation 6

Existing techniques depend on manual input specification, which is labor intensive and tedious.

Open-Problem 3 (Technical)

Although automated input type identification techniques exist, they are applicable to only limited interfaces and are BSP. Leaving a need to develop techniques that perform inference for type 2 or type 3 bootloaders.

Interface Invocation. Bootloaders expose diverse interfaces for both runtime and post-boot interactions, which serve as key entry points for dynamic analysis (§ 3). However, these interfaces are often poorly documented, dynamically configured, and vary across platforms. For example, in EDK-II, protocol drivers (sas_4) perform a dynamic check (`DriverBindingSupported`) before loading to ensure the required hardware is present [148]. Even when compiled, some protocols may not be active at runtime, highlighting the need for *interface identification* [22]. Once active interfaces are identified, the next step is *interface triggering*. In Listing 1, some interfaces, like `CreateKeyHandler` (sas_3), are externally exposed and can be invoked directly, while others, such as certain SMM protocols (`RngProtocol`), require firmware modifications to be accessible.

Interfaces might depend on platform-specific initialization order and hardware state. Techniques must replicate these dependencies (*e.g.,* NVRAM state, previously loaded stages) to exercise target interfaces effectively, which we call *dependency identification*. Failure to account for these prerequisites can prevent the execution of critical functions

Table 6: Dynamic analysis capabilities for different attack surfaces by bootloader type (1, 2, 3). Automated techniques are marked ⚙; manual are marked M; ✖ means no existing technique. Tool names can be cross-referenced in Table 3. Parenthetical counts denote known vulnerabilities per surface.

	Necessary Capabilities	Techniques	HW Surfaces		SW Surfaces			
			has_1(2)	has_2(4)	sas_1(12)	sas_2(16)	sas_3(10)	sas_4(45)
Input Generation	Argument Type Identification	Type Inference	✖	1:⚙ [55]	1:⚙ [55], [165]	1:⚙ [165]	1:⚙ [55], [165], [170]	1:⚙ [55], [165]
		Grammar Specification	1:M [167] 3:M [53]	1:M [122], [167] 2:M [51], [156] 3:M [53], [156]	✖	1:⚙ [95] M [112], [167]	1:⚙ [95], [137] M [122], [167] 2:M [51], [156] 3:M [156]	1:M [122], [124], [167] 2:M [51], [62], [156] 3:M [156]
	Argument Value Identification	Inference	✖	1:⚙ [55]	1:⚙ [55]	✖	1:⚙ [55], [170]	1:⚙ [55]
		Fixed / Random Values	1:M [167] 3:M [53]	1:M [122], [167] 2:M [51], [156] 3:M [53], [156]	✖	1:⚙ [95], [165] M [112], [167]	1:⚙ [95], [137] M [122], [167] 2:M [51], [156] 3:M [156]	1:M [122], [124], [167] 2:M [51], [62], [156] 3:M [156]
	Interface Invocation	Harnessing / Hooking	1:M [167] 3:M [53]	1:⚙ [55] M [122], [167] 2:M [51], [156] 3:M [53], [156]	1:⚙ [55], [165]	1:⚙ [95], [165] M [112], [167]	1:⚙ [55], [95], [137], [165], [170] M [122], [167] 2:M [51], [156] 3:M [156]	1:⚙ [55], [165] M [122], [124], [167] 2:M [51], [62], [156] 3:M [156]
	Target Execution	Off-Shelf Simulator	1:⚙ [55], [112], [122], [137], [165], [167], [170],M [124] 2:⚙ [51],M [156] 3:⚙ [53],M [156]					
		Automated Rehosting	✖					
		On-Device	1:⚙ [95] 2:M [62] 3:✖					
	Invalid State Detection	Crash / Hang Detection	1:⚙ [55], [112], [122], [124], [137], [165], [167], [170] 2:⚙ [51], [62], [156] 3:⚙ [53], [156]					
		Sanitizer	1:⚙ [55], [112], [122], [165] 2:⚙ [51], [156] 3:⚙ [53], [156]					
		Heuristic Rules	1:⚙ [95], [122], [170] 2,3:✖					
	Feedback	Coverage / Tracing	1:⚙ [55], [112], [122], [124], [165], [167], [170],M [95], [137] 2:⚙ [51], [156] 3:⚙ [53], [156]					
		Taint / Dataflow	1:⚙ [170] 2,3:✖					
		Manual Guidance	1:M [124], [137] 2:M [62] 3:✖					

or cause false negatives. For instance, GRUB uses `grub.cfg` to enforce deterministic boot sequences and ensure system support before proceeding to later stages.

Table 6 summarizes approaches across attack surfaces. For *interface identification*, manual inspection of source code or debug logs remains the most common method, with automated discovery rarely applied and limited to select UEFI applications and protocols [55], [165], [170]. For *interface triggering*, both manual (**M**) and automated (⚙) harness generations techniques exist. Manual approaches dominate across all attack surfaces, especially has_1 and has_2, whereas automation has been demonstrated only in a few cases (sas_1, sas_3, sas_4).

The most effective automated techniques combine type-aware fuzzing or input generation with automated harnessing, allowing deeper exploration and uncovering more vulnerabilities than manual methods.

Challenges and coverage vary by bootloader family. Type 2 bootloaders are highly scriptable and configuration-driven, making interface discovery and deterministic replay comparatively straightforward (high coverage for post-boot interfaces). While type 3 are board- and peripheral-centric: hardware initialization and board-specific drivers make interfaces highly platform dependent and harder to automate. Conversely, type 1 are the most complex: dynamic driver binding, multiple privilege modes (including SMM), and deeply split runtime vs. pre-boot functionality mean many interfaces are transient or inaccessible without firmware modification. As a result, most automated techniques focus on post-boot and exposed type 1 bootloader applications. Early boot and platform-specific interfaces (network stacks, low-level device enumerations, SMM-only protocols) receive far less automated coverage and often require manual or platform-specific engineering to test.

Observation 7

Interface discovery and invocation are bottlenecks for dynamic analysis. Identification, triggering, and dependency reconstruction are largely manual across bootloaders.

Invalid State Detection and Feedback. Dynamic analysis of bootloaders must be able to detect when the system reaches an invalid or buggy state. Unlike general applications or OS kernels, which benefit from virtual memory protections that make memory corruption bugs manifest as observable crashes, bootloaders, especially in early stages (*i.e.,* stages 1–3), lack such protections. Even in later stages, virtual memory may be uninitialized depending on the bootloader, so memory safety bugs often go undetected, a situation common in deeply embedded software [103].

A common solution is to use a sanitizer (*e.g.,* Address Sanitizer (ASan) [127], Undefined Behavior Sanitizer (UBSan) [23]). Sanitizers increase bug detection through instrumentation, but they rely on virtual memory protections [132], even the bare-metal KASAN implementation results into similar limitations [111]. Bootloaders also face memory constraints that complicate sanitizer integration. For example, standard U-Boot systems provide enough contiguous memory to accommodate ASan (*e.g.,* `0x00002000 - 0x00fbff20` [32]), but handling shadow and poisoned memory before permanent memory initialization in stage 2 (Figure 1) presents an engineering challenge.

As shown in Table 6, three detection techniques are used: *crash and hang detection* (widely used but misses

early-stage violations), *sanitizers* (strong but rarely deployed due to hardware constraints), and *heuristic rules* (semi-automated and bug-specific).

Although few techniques use sanitizers, these are extremely customized for specific use-cases and attack surfaces. For example, FuzzUEr [55] had to heavily integrate a custom adaption of ASan [127] to work only on stage 3.

Observation 8

Lightweight, general-purpose sanitizers that operate without hardware protections are currently lacking.

Open-Problem 4 (Engineering)

Designing runtime checks that provide strong bug oracles while respecting bootloader memory and initialization constraints remains an open challenge.

Feedback mechanisms are also critical for dynamic analysis, particularly for fuzzing. Most tools rely on *coverage and tracing* feedback to guide exploration [55], [112], [122], [124], [165], [167], [170], while advanced feedback like *taint or dataflow tracking* has been demonstrated in limited settings [170]. Some tools use *manual guidance* when automated feedback is unavailable, though this approach does not scale [62], [95], [137].

4.5.2. Tools Evaluation

Our selection methodology (§ 4.3) resulted in 4 tools, which we evaluated using our BOOTBENCH dataset.

Results Of the tools evaluated, only FuzzUEr [55] detected any injected vulnerabilities, finding 1 of 16 (6%), a memory safety bug. FuzzUEr missed others because it does not support the relevant attack surfaces and could not reconstruct the state required to reach deeper code paths.

Other tools, including HBFA [122], BootFuzz [137], and efi_fuzz [112], *failed to detect any vulnerabilities for similar reasons*. BootFuzz is a simple fuzzer that passes random values without type awareness and relies on manual feedback, which limits its effectiveness.

HBFA uses stubs and custom harnesses to execute drivers in userspace, enabling sanitizers and fuzzers like AFL. However, this approach sacrifices significant functionality because many code paths are stubbed out. For example, the `USB_IO_PROTOCOL` is fully stubbed. The drivers depending on it still compile, but most of the original functionality is removed, as the stubs simply return `EFI_SUCCESS`. Consequently, HBFA's flexibility comes at the cost of missing realistic execution paths and potential vulnerabilities.

efi_fuzz focuses exclusively on NVRAM variable fuzzing (sas_2), which represents one of the most constrained attack surfaces, further restricting its effectiveness.

Dynamic analysis tools exhibited high false negative rates overall, largely due to incomplete interface identification and triggering across architectures. FuzzUEr achieved the lowest false negative rate by leveraging type-aware fuzzing and input mutation, while others failed to explore runtime paths effectively. Memory-safety bugs were most likely to be discovered, whereas logic and configuration vulnerabilities consistently went undetected. Although dynamic tools produced fewer false positives than static analysis, their high false negative rates reveal gaps in runtime state reconstruction and dependency modeling.

Finding 6

Existing dynamic analysis tools are specific to one type of bootloader and focus on a single attack surface, resulting in low coverage. Except for FuzzUEr (which found one), none of the tools were able to find any vulnerabilities in BOOTBENCH. Memory-safety issues are partially detectable, while state-dependent and configuration vulnerabilities remain beyond current capabilities.

4.5.3. Adoption in Bootloaders

Similar to static analysis, we checked whether open-source bootloaders use dynamic techniques, including OSS-Fuzz integration.

Results. Only EDK-II and systemd-boot use OSS-Fuzz [142], [148], achieving 25.72% and 29.37% code coverage respectively—low figures reflecting the use of general-purpose rather than bootloader-specific fuzzing.

4.6. Attack Surface Coverage

Comprehensive vulnerability detection requires reasoning across two dimensions of bootloader security: *attack surfaces* and *execution stages*. Attack surfaces define *where* vulnerabilities can be triggered, while boot stages define *when* they manifest. Most existing frameworks focus on only one axis, either analyzing software-facing interfaces or symbolically executing a specific stage, without spanning both, as summarized in Table 3. Consequently, even tools with advanced symbolic reasoning miss hardware-induced flaws and state persistence across stages. Coverage of sas_4 (boot-time interfaces) dominates prior work, while early-stage hardware and late-stage runtime surfaces remain largely unexplored. For instance, as shown in Table 6, despite the prevalence of vulnerabilities through sas_2, only a limited number of works explore it. Specifically, we found over 40 vulnerabilities (in our dataset) along attack surfaces that are covered by none of the techniques. Many of these require cross-surface and cross-stage reasoning. Addressing this requires extending current frameworks to jointly analyze vulnerabilities across both axes, enabling consistent reasoning from early firmware initialization through runtime services.

Open Problem 5 (Technical)

Current frameworks cannot jointly model vulnerabilities across attack surfaces *and* boot stages, hindering detection of flaws that traverse initialization boundaries or evolve from hardware faults to software logic errors.

5. Defences

We examine three complementary defence mechanisms: exploit mitigation to limit exploitability (§5.1), security

hardening to reduce the attack surface (§5.2), and vulnerability prevention to eliminate bug classes at the source (§5.3).

5.1. Exploit Mitigation

Exploit mitigation mechanisms do not eliminate vulnerabilities but significantly raise the difficulty of exploitation, reducing the likelihood of converting memory errors into control-flow hijacks. Specifically, we checked for six common mitigations across all bootloaders in our dataset, *i.e.,* Address Space Layout Randomization (ASLR) [43], Stack Canaries [30], Non-Executable Memory (NX) memory [14], Pointer Athentication Codes (PAC) [97], Control-Flow Integrity (CFI) [2], and Relocation Read-Only (RELRO) [29]. We performed build system review (for open-source bootloaders), `checksec` [73] analysis and binary inspection for closed-source (binary-only) bootloaders to check if any of the mitigations are enabled. As summarized in Table 8, adoption is sparse due to early-stage hardware constraints, static linking, and the absence of runtime memory management [144], [149]. Most mitigations rely on an MMU, entropy sources, or virtual-memory-based compiler instrumentation, features rarely available in early boot stages [68], [144], [149].

Despite these challenges, several mitigations show potential for incremental adoption. NX enforcement, Stack Canaries, and PAC have emerged as the most practical defenses, as they can be enabled once the MMU and secure key material are available [14], [30], [79], [117]. ASLR, CFI, and RELRO remain uncommon due to strict initialization ordering, lack of relocatable code, or incomplete toolchain support [52], [86], [110], [168]. In general, effective mitigation deployment requires memory protection hardware, deterministic stage sequencing, and compiler integration for consistent instrumentation. A detailed per-mitigation analysis, including design constraints and feasibility by boot stage, is provided in Appendix C.

Observation 9

Most bootloaders lack exploit mitigations due to hardware and memory constraints. Among the feasible options, NX, PAC, and Stack Canaries offer the most practical protection once memory management and key initialization become available, while ASLR, CFI, and RELRO remain limited by toolchain and early boot dependencies.

Open-Problem 6 (Engineering)

Current exploit mitigations prioritize performance and use OS primitives. Bootloaders, where performance is less critical, need new mitigations that trade speed for independence from OS features.

5.2. Security Hardening

Security hardening techniques aim to reduce the attack surface and limit the impact of exploitation attempts. Unlike prevention methods, which eliminate flaws, hardening constrains system behavior through runtime enforcement.

5.2.1. Generic Techniques

Common defenses such as Privilege Separation [15], [56] and Memory Region Protection [149], [161] are well-studied in operating systems but rarely applied to bootloaders. Most bootloaders execute monolithically at the highest privilege level (*e.g.,* x86 ring 0 or ARM EL3/EL2), offering no isolation between components. Partial separation exists in UEFI (*i.e.,* SMM) and ARM Trusted Firmware (*i.e.,* TrustZone) [6], [47], but a single module compromise still endangers the entire chain [169].

Memory region protection similarly depends on MMU support. While UEFI can enforce page permissions (*e.g.,* NX stacks) once virtual memory is active, many bootloaders rely on flat mappings or lack fine-grained control [6], [148]. Hardware-backed partitioning (*e.g.,* secure memory regions) remains coarse and difficult to generalize.

Recent work such as DECAF [22] shows that automated debloating can remove up to 70% of UEFI code without breaking functionality, suggesting that targeted reduction may complement coarse-grained protection.

Observation 11

Bootloader hardening remains coarse-grained due to early-stage execution constraints. Privilege separation is limited to special modes, and memory protection is often static or disabled before full MMU initialization.

5.2.2. Bootloader-Specific Techniques

Bootloaders also employ dedicated integrity mechanisms unavailable at the OS level: Early Launch Anti-Malware (ELAM) verifies kernel drivers in Windows Boot Manager [159], Secure Boot enforces signed execution across stages [31], [42], and Measured Boot extends TPM-based integrity checks for attestation [152], [160]. These measures form the foundation of the modern firmware trust chain.

Secure and measured boot depend on reliable key management and TPM integration [107], [150], while ELAM is tied to the Windows ecosystem. Hardware diversity, limited memory, and cryptographic overhead restrict broader use [147], [162]. Interface lockdowns or signature checks also complicate debugging and firmware updates [37], [146].

Observation 12

BSP hardening is rarely used. Secure and measured boot dominate, while runtime defenses like privilege separation or fine-grained memory control remain impractical under current hardware and toolchain constraints.

5.3. Vulnerability Prevention

Vulnerability prevention aims to eliminate bug classes before they manifest at runtime. Two existing approaches are investigated.

5.3.1. Language-Based Memory Safety Retrofitting

These techniques retrofit safety into C/C++ code by tracking pointer metadata and enforcing bounds at compile time or runtime. Examples include CETS [106], Checked

C [39], and related pointer-tracking systems such as LowFat, SoftBound, and FatPointers [74], [105], [177]. They are generally implemented via compiler instrumentation (*e.g.,* CLANG [75]) and can enforce spatial and temporal safety without rewriting large codebases.

However, their adoption in bootloaders is limited. Lack of virtual memory and tight memory budgets make these techniques difficult to deploy. For instance, LowFat Pointers rely on virtual address partitioning [74], while CETS and SoftBound impose runtime and memory overheads exceeding 100% [105], [106]. Checked C's incremental annotations have been partially adapted to EDK-II [21], but only 86% of pointers could be instrumented. Bootloaders' statically linked, hardware-specific layouts further hinder direct adoption.

Finding 7

Bootloaders lack systematic language-based memory safety retrofitting. Hardware constraints, static layouts, and minimal runtime support present opportunities for specialized, domain-aware approaches.

5.3.2. Memory-Safe Languages

Memory-safe languages (*e.g.,* Java, Go, Rust) prevent common memory errors through compile-time enforcement. Rust is the most practical choice for low-level systems due to its ownership and lifetime model [89]. Nevertheless, adoption in bootloaders remains rare: only two open-source examples use Rust [108], [113] and one uses Go [80]. Rust-EDK-II [114] enables UEFI applications in Rust but not the bootloader core. As observed by Sharma *et al.,* [129], Rust's runtime and tooling remain immature for boot environments.

Observation 10

Vulnerability prevention techniques require significant adaptation for bootloaders. Even mature compiler- or language-based methods see minimal use due to limited memory, hardware dependencies, and legacy toolchains.

6. Discussion

Our study highlights that while bootloader security has recently gained attention, existing research remains fragmented across attack surfaces, analysis techniques, and mitigation strategies. We consolidate our **findings**, **observations**, and **open problems** to identify key challenges and outline future research directions.

Fragmented Attack Surface Understanding. Bootloaders exhibit diverse and evolving attack surfaces (**Observations 1, 2, 7, 12**), yet most prior works focus narrowly on boot-time or post-boot interfaces. **Finding 1** shows that attack surfaces are not consistent across bootloaders, while **Finding 2** indicates that several important surfaces remain underexplored. This fragmentation limits reproducibility and obscures inter-stage vulnerabilities. Future work should develop systematic methods to automatically discover, characterize, and standardize bootloader interfaces across hardware and firmware implementations.

Open-Problem 7 (Technical)

How can we automatically and thoroughly identify bootloader attack surfaces across different hardware and execution stages, including early-boot interfaces?

Limitations of Static and Dynamic Analyses. Existing analysis tools face fundamental limitations (**Findings 3–6**, **Observations 4–8**). Static analyses struggle with cross-language reasoning (C/ASM), limited taint propagation, and missing BSP or interrupt models. Dynamic analyses often depend on incomplete rehosting setups and manual input specifications (**Observations 5–6**). **Finding 4** shows that asynchronous and interrupt-driven control flows exacerbate these issues. Hybrid techniques that combine symbolic reasoning with emulator-assisted execution may improve accuracy and coverage (**Open-Problems 2-3**).

Inter-Stage Data and Cross-Phase Vulnerabilities. Bootloaders exchange persistent state across stages (**Observations 2, 6**), yet current tools treat stages in isolation. Whole-chain reasoning, tracking control and data flow from early firmware through the OS handoff, remains largely unaddressed. Symbolically modeling NVRAM and persistent storage, or using staged symbolic execution, could expose vulnerabilities that span multiple phases. However, our evaluation also revealed a deeper limitation: even when vulnerabilities were injected across all six attack surfaces, existing tools only detected issues in sas_1 and sas_4. This demonstrates that current frameworks cannot simultaneously reason *across attack surfaces* and *across boot stages*, leaving significant blind spots in end-to-end coverage (**Open-Problem 5**).

Runtime Analysis and Lightweight Instrumentation. **Finding 5** and **Observation 8** reveal that existing sanitizers and runtime tools assume OS-level support and fail under bootloader constraints. Developing self-contained, lightweight runtime checkers that operate without dynamic memory allocators or interrupts remains a key challenge (**Open-Problem 4**).

Gaps in Exploit Mitigation and Prevention. Our analysis (**Findings 6–7**, **Observations 9–12**) shows that most bootloaders lack exploit mitigations due to hardware constraints and the absence of privilege separation. While lightweight options such as NX, PAC, and Stack Canaries are feasible, other mitigations (ASLR, CFI, RELRO) are often unavailable. Moreover, **Finding 7** highlights the lack of systematic approaches for memory-safety retrofitting, and **Observation 11–12** underscores the dependence on OS primitives. We need to develop domain-specific exploit mitigations and memory-safety retrofitting techniques tailored for early-stage environments, balancing determinism, verifiability, and security (**Open-Problem 6**).

7. Related Work

Several works systematize components of the boot process, but none focus solely on bootloaders.

Recent works [9], [140] address security concerns in type 1 bootloaders (*i.e.,* firmware), examining the UEFI boot process, system security, and current attack vectors.

Most existing SoKs that involve bootloaders focus on hardware security mechanisms [18], [135], [155], [173], [176], particularly hardware isolation environments. Zhang *et al.,* categorize hardware isolation environments and their security applications [173], while others address architecture-specific isolation such as TrustZone [5], [18] or Intel SGX [155]. Fasano *et al.,* taxonomize embedded system rehosting [44], and Jain *et al.,* address virtual machine introspection and the semantic gap [72].

8. Threats For Validity

The data collected and presented (*i.e.,* § 2, § 3, § 4, § 5.1, § 5.3, § 5.2) is accurate based on our analysis; however, companies may use in-house tools or bootloaders that we are unaware of and that may contradict our findings. Additionally, dataset biases may arise from the keywords used to collect CVEs and vulnerability-related commit messages, which were manually tuned through iterative analysis.

9. Conclusion

Bootloader security has become increasingly important due to a surge in high-profile vulnerabilities across all bootloader types, alongside a rise in vulnerability detection tools. However, many of these tools remain ineffective, often overlooking critical attack surfaces. While detection capabilities have improved, there has not been a corresponding increase in efforts around security hardening, exploit mitigation, or vulnerability prevention. Through the findings of this work, we aim to guide future efforts toward more comprehensive and effective approaches to securing and understanding bootloaders.

Ethical Considerations

All the vulnerabilities we collected as part of the paper are publicly available, and we do not introduce any additional risk. There are no new vulnerabilities identified by the existing tools.

Acknowledgments

We would like to thank the shepherd and reviewers for their valuable comments, which greatly improved our paper. This research was supported by the Defense Advanced Research Projects Agency (DARPA) under contract number N660012224037, and the National Science Foundation (NSF) under Grants CNS-2340548 and TIP-2534021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the DARPA, the NSF, or the U.S. Government.

References

[1] Sok: All you ever wanted to know about bootloader security but were afraid to ask extended report. https://drive.google.com/file/d/1dOYmtbIVnf6QyqCCQo-RjjKsIwKrhl7X/view?usp=sharing, 2025.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, page 340–353, New York, NY, USA, 2005. Association for Computing Machinery.

[3] Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Arnaud Blouin, Djamel Eddine Khelladi, and Jean-Marc Jézéquel. *Learning from thousands of build failures of Linux kernel configurations*. PhD thesis, Inria; IRISA, 2019.

[4] Android. Bootloader overview. https://source.android.com/docs/core/architecture/bootloader, 2021.

[5] ARM. ARM security technology building a secure system using TrustZone technology. https://developer.arm.com/documentation/PRD29-GENC-009492/latest/, 2009.

[6] ARM. Trusted firmware-a. https://github.com/ARM-software/arm-trusted-firmware, 2013.

[7] barebox. barebox. https://github.com/barebox/barebox, 2005.

[8] Michael Bartock, Murugiah Souppaya, Ryan Savino, Tim Knoll, Uttam Shetty, Mourad Cherfaoui, Raghu Yeluri, Akash Malhotra, Don Banks, Michael Jordan, Dimitrios Pendarakis, J R Rao, Peter Romness, and Karen Scarfone. Hardware-enabled security : enabling a layered approach to platform security for cloud and edge computing use cases. Technical Report NIST IR 8320, National Institute of Standards and Technology (U.S.), Gaithersburg, MD, May 2022.

[9] Vladimir Bashun, Anton Sergeev, Victor Minchenkov, and Alexandr Yakovlev. Too young to be secure: Analysis of UEFI threats and vulnerabilities. In *14th Conference of Open Innovation Association FRUCT*, pages 16–24. IEEE.

[10] O. Bazhaniuk, J. Loucaides, L. Rosenbaum, M. Tuttle, and V. Zimmer. Symbolic execution for bios security. *Workshop on Offensive Technologies*, 2015.

[11] Binarly. Binary risk hunt. https://risk.binarly.io/, 2025.

[12] Inc. Black Duck Software. Coverity scan static analysis. https://scan.coverity.com/, 2024.

[13] Slim Bootloader. slimbootloader. https://github.com/slimbootloader/slimbootloader, 2018.

[14] Shlomi Boutnaru. Security — nx bit (non-executable). https://medium.com/@boutnaru/security-nx-bit-non-executable-18759fd2802e.

[15] David Brumley and Dawn Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[16] D. Brylow, N. Damgaard, and J. Palsberg. Static checking of interrupt-driven software. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 47–56, 2001.

[17] Samuele Buro, Roy L. Crole, and Isabella Mastroeni. On Multi-language Abstraction. In David Pichardie and Mihaela Sighireanu, editors, *Static Analysis*, pages 310–332, Cham, 2020. Springer International Publishing.

[18] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. SoK: Understanding the prevailing security vulnerabilities in TrustZone-assisted TEE systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1432. IEEE.

[19] Charu Chauba. The architecture of vmware esxi. https://microage.com/wp-content/uploads/2016/02/ESXi_architecture.pdf.

[20] Sourag Cherupattamoolayil, Arunkumar Bhattar, Connor Everett Glosner, and Aravind Machiry. Adding spatial memory safety to edk ii through checked c (experience paper). *Proc. ACM Softw. Eng.*, 2(ISSTA), June 2025.

[21] Sourag Cherupattamoolayil, Arunkumar Bhattar, Connor Everett Glosner, and Aravind Machiry. Adding spatial memory safety to edk ii through checked c (experience paper). *Proc. ACM Softw. Eng.*, 2(ISSTA), June 2025.

[22] Jake Christensen, Ionut Mugurel Anghel, Rob Taglang, Mihai Chiroiu, and Radu Sion. DECAF: Automatic, adaptive de-bloating and hardening of COTS firmware. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1713–1730. USENIX Association, August 2020.

[23] Clang Community. Undefinedbehaviorsanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2025.

[24] coreboot. coreboot. https://github.com/coreboot/coreboot, 2003.

[25] coreboot. seabios. https://github.com/coreboot/seabios, 2008.

[26] coreboot project. coreboot documentation. https://doc.coreboot.org/, 2024.

[27] coreboot Project. AMD Platform Security Processor (PSP) Firmware Integration Guide, 2025.

[28] The MITRE Corporation. Cve: Common vulnerabilities and exposures. https://www.cve.org/, 2025.

[29] CTF101. Relocation read-only (relro). https://ctf101.org/binary-exploitation/relocation-read-only/.

[30] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, page 555–566, New York, NY, USA, 2015. Association for Computing Machinery.

[31] Windows Hardware Developers. Secure boot. https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-secure-boot.

[32] The U-Boot development community. The u-boot documentation. https://docs.u-boot.org/en/latest/.

[33] Google Devices. Cve-2023-48426 detail. https://nvd.nist.gov/vuln/detail/CVE-2023-48426, 2023.

[34] Eclypsium. There's a hole in the boot. https://eclypsium.com/blog/theres-a-hole-in-the-boot/, 2020.

[35] Eclypsium. Trickbot now offers "trickboot": Persist, brick, profit. https://eclypsium.com/blog/trickbot-now-offers-trickboot-persist-brick-profit/, 2020.

[36] Eclypsium. The real shim shady - how cve-2023-40547 impacts most linux systems. https://eclypsium.com/blog/the-real-shim-shady-how-cve-2023-40547-impacts- most-linux-systems/, 2024.

[37] Eclypsium Research Team. Enhanced threat detection: Bootloaders, bootkits, and secure boot. 2025.

[38] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. An empirical study of configuration mismatches in linux. In *Proceedings of the 21st International Systems and Software Product Line Conference-Volume A*, pages 19–28, 2017.

[39] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 53–60, September 2018.

[40] EMBA. Emba: The security analyzer for firmware of embedded devices. https://github.com/e-m-b-a/emba, 2020.

[41] Jakob Engblom. Finding bios vulnerabilities with symbolic execution and virtual platforms, 2019.

[42] Erdem. What's the difference between secure boot and measured boot? https://community.juniper.net/blogs/elevate-member/2020/12/22/whats-the-difference-between-secure-boot-and-measured-boot.

[43] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[44] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, Davide Balzarotti, and William Robertson. SoK: Enabling security analyses of embedded systems via rehosting. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 687–701. ACM.

[45] Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. How the ELF ruined christmas. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 643–658, Washington, D.C., August 2015. USENIX Association.

[46] Andrea Flexeder, Bogdan Mihaila, Michael Petter, and Helmut Seidl. Interprocedural Control Flow Reconstruction. In Kazunori Ueda, editor, *Programming Languages and Systems*, pages 188–203, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[47] Unified EFI Forum. Unified extensible firmware interface (uefi) specification. https://uefi.org/specifications, 2024.

[48] Open-Source Firmware Foundation. Open source firmware projects. https://opensourcefirmware.foundation/projects/, 2025.

[49] Iván Arce Francisco Falcon. Pixiefail: Nine vulnerabilities in tianocore's edk ii ipv6 network stack. https://blog.quarkslab.com/pixiefail-nine-vulnerabilities-in-tianocores-edk-ii -ipv6-network-stack.html, 2024.

[50] Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgeny Groshev. Configfix: Interactive configuration conflict resolution for the linux kernel. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 91–100. IEEE, 2021.

[51] Leonardo Galli. Emmulater: Fuzzing the ios boot loader. https://github.com/galli-leo/emmutaler/blob/master/docs/thesis.pdf, 2021.

[52] Jonathan Ganz and Sean Peisert. Aslr: How robust is the randomness? In *2017 IEEE Cybersecurity Development (SecDev)*, pages 34–41, 2017.

[53] Patrick Gersch. Aspfuzz: Fuzzing the amd secure processors rom bootloader with libafl using qemu full-system emulation. https://github.com/TeumessianFox/ASPFuzz, 2023.

[54] GitHub. Codeql. https://github.com/github/codeql, 2018.

[55] Connor Glosner and Aravind Machiry. FUZZUER: Enabling Fuzzing of UEFI Interfaces on EDK-2. In *Proceedings 2025 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2025. Internet Society.

[56] GNU. Gnu grub manual 2.12: Security. https://www.gnu.org/software/grub/manual/grub/html_node/Security.html.

[57] GNU. Grub. https://git.savannah.gnu.org/git/grub, 2009.

[58] GNU. the gnu grub developer manual. https://www.gnu.org/software/grub/manual/grub-dev/grub-dev.pdf, 2023.

[59] Google. Depthcharge. https://chromium.googlesource.com/chromiumos/platform/depthcharge/, 2012.

[60] Grandidierite. Bios interrupts. https://grandidierite.github.io, 2018.

[61] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589, 2014.

[62] Roee Hay, Aleph Research, and HCL Technologies. fastboot oem vuln: Android Bootloader Vulnerabilities in Vendor Customizations. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[63] Susan Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, January 1997.

[64] International Business Machine (IBM). *Personal System/2 and Personal Computer BIOS Interface Technical Reference*. International Business Machine (IBM), 1987.

[65] Red Hat Inc. Cve-2023-40547 detail. https://nvd.nist.gov/vuln/detail/cve-2023-40547, 2023.

[66] Intel. A technical look at intel® control-flow enforcement technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement- technology.html.

[67] Intel. Tsffs. https://github.com/intel/tsffs/, 2023.

[68] Intel. Control-flow enforcement technology (cet) shadow stack. https://docs.kernel.org/arch/x86/shstk.html, 2025.

[69] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*, 2024.

[70] Intel Corporation. Boot Guard Technology, 2025.

[71] Microsoft Threat Intelligence. Analyzing open-source bootloaders: Finding vulnerabilities faster with ai. https://www.microsoft.com/en-us/security/blog/2025/03/31/analyzing-open-source-bootloaders-finding- vulnerabilities-faster-with-ai/, 2025.

[72] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter, and Radu Sion. SoK: Introspections on trust and the semantic gap. In *2014 IEEE Symposium on Security and Privacy*, pages 605–620. ISSN: 2375-1207.

[73] Tobias Klein and Brian Davis. checksec. https://github.com/slimm609/checksec.git, 2013.

[74] Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, and Andre DeHon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, page 721–732, New York, NY, USA, 2013. Association for Computing Machinery.

[75] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[76] Mark Lechtik, Vasily Berdnikov, Denis Legezo, and Ilya Borisov. Moonbounce: the dark side of uefi firmware. https://securelist.com/moonbounce-the-dark-side-of-uefi-firmware/105468/, 2022.

[77] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. Path-sensitive and alias-aware typestate analysis for detecting os bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 859–872, New York, NY, USA, 2022. Association for Computing Machinery.

[78] Ziyang Li, Saikat Dutta, and Mayur Naik. Iris: Llm-assisted static analysis for detecting security vulnerabilities, 2025.

[79] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, Santa Clara, CA, August 2019. USENIX Association.

[80] linuxboot. linuxboot. https://github.com/linuxboot/linuxboot, 2016.

[81] Yin Liu and Ana Milanova. Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 146–155, 2010.

[82] Startup Defense LLC. Cold boot attacks: Understanding data theft in ram. https://www.startupdefense.io/cyberattacks/cold-boot-attack.

[83] Lenovo Group Ltd. Cve-2024-7756 detail. https://www.cve.org/CVERecord?id=CVE-2024-7756, 2024.

[84] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. Spider: Enabling fast patch propagation in related software repositories. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1562–1579. IEEE, 2020.

[85] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[86] Stephen Marz. Dynamic linking. *LWN.net*, 2020.

[87] Alex Matrosov. The untold story of the blacklotus uefi bootkit. https://www.binarly.io/posts/The_Untold_Story_of_the_BlackLotus_UEFI_Bootkit/index.html, 2023.

[88] Alex Matrosov, Eugene Rodionov, , and Sergey Bratus. *Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats*. No Starch Press, 2019.

[89] Nicholas D. Matsakis and Felix S. Klock. The rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery.

[90] mcu tools. mcuboot. https://github.com/mcu-tools/mcuboot.

[91] mcu tools. mcuboot. https://docs.mcuboot.com/.

[92] Zephyr Project members and individual contributors. Zephyr project documentation. https://docs.zephyrproject.org/latest/.

[93] Trend Micro. Insecurity despite obscurity: Thunderstrike 2 rootkit can now infect macs remotely. https://www.trendmicro.com/vinfo/us/security/news/ vulnerabilities-and-exploits/thunderstrike-2-rootkit-can-now-infect- macs-remotely, 2015.

[94] Microsoft. Hyper-v technology overview. https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-overview?pivots=windows, 2025.

[95] Microsoft. Uefi scanning in defender for endpoint. https://learn.microsoft.com/en-us/defender-endpoint/uefi-scanning-in-defender-for-endpoint, 2025.

[96] MITRE. Common weakness enumeration. https://cwe.mitre.org/data/downloads.html.

[97] MITRE. Pointer authentication. https://d3fend.mitre.org/technique/d3f:PointerAuthentication/.

[98] MITRE. Cve-2018-18439 detail. https://nvd.nist.gov/vuln/detail/cve-2018-18439, 2018.

[99] MITRE. Finfisher. https://attack.mitre.org/software/S0182/, 2018.

[100] MITRE. Cve-2019-13104 detail. https://nvd.nist.gov/vuln/detail/CVE-2019-13104, 2019.

[101] MITRE. Cve-2022-44371 detail. https://nvd.nist.gov/vuln/detail/CVE-2022-44371, 2022.

[102] MITRE. Cve-2024-36435 detail. https://nvd.nist.gov/vuln/detail/cve-2024-36435, 2024.

[103] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*, 2018.

[104] Sarah Nadi and Ric Holt. The linux kernel: A case study of build system variability. *Journal of Software: Evolution and Process*, 26(8):730–746, 2014.

[105] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. *SIGPLAN Not.*, 44(6):245–258, June 2009.

[106] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. *SIGPLAN Not.*, 45(8):31–40, June 2010.

[107] Duc Nguyen, Miroslaw Staron, and Mark Fiedler. Enhancing iot device security in autonomous building systems. *Halmstad University Institutional Repository*, 2024.

[108] nihalpasham. rustboot. https://github.com/nihalpasham/rustBoot/, 2021.

[109] ANDERSEN L. O. Program analysis and specialization for the c programming language. *Ph. D. Thesis*, 1994.

[110] Trail of Bits. Let's talk about cfi: Clang edition. https://blog.trailofbits.com/2016/10/17/lets-talk-about-cfi-clang-edition/, 2016. Accessed: 2025-08-15.

[111] Android Offsec. Bare-metal KASan implementation.

[112] Sentinel One. efi_fuzz. https://github.com/Sentinel-One/efi_fuzz, 2020.

[113] Rust OSDev. bootloader. https://github.com/rust-osdev/bootloader, 2017.

[114] Rust OSDev. uefi-rs: Rusty wrapper for the unified extensible firmware interface. https://github.com/rust-osdev/uefi-rs, 2017.

[115] Niels Pfau and Patrick Kochberger. Analysis of the windows control flow guard. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, ARES '24, New York, NY, USA, 2024. Association for Computing Machinery.

[116] QEMU. Qemu: A generic and open source machine emulator and virtualizer. https://www.qemu.org/, 2024.

[117] Qualcomm Technologies. Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions, 2017.

[118] Inc. Red Hat. Cve-2020-10713 detail. https://www.cve.org/CVERecord?id=CVE-2020-10713, 2020.

[119] Inc. Red Hat. Cve-2020-25647 detail. https://nvd.nist.gov/vuln/detail/CVE-2020-25647, 2020.

[120] Inc. Red Hat. Cve-2024-40547 detail. https://nvd.nist.gov/vuln/detail/CVE-2023-40547, 2023.

[121] Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. BootStomp: On the security of bootloaders in mobile devices. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[122] Brian Richardson, Chris Wu, Jiewen Yao, and Vincent J. Zimmer. Using host-based firmware analysis to improve platform resiliency. 2019.

[123] Xavier Rival and Kwangkeun Yi. Introduction to static analysis, 2020.

[124] Ryan. uefi_fuzzer. https://github.com/oscardagrach/uefi_fuzzer, 2023.

[125] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[126] SeaBIOS. Seabios. https://seabios.org, 2015.

[127] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012.

[128] Md Shafiuzzaman, Achintya Desai, Laboni Sarker, and Tevfik Bultan. STASE: Static Analysis Guided Symbolic Execution for UEFI Vulnerability Signature Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1783–1794, Sacramento CA USA, October 2024. ACM.

[129] Ayushi Sharma, Shashank Sharma, Sai Ritvik Tanksalkar, Santiago Torres-Arias, and Aravind Machiry. Rust for embedded systems: Current state and open problems. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 2296–2310, New York, NY, USA, 2024. Association for Computing Machinery.

[130] Mingjie Shen, Akul Pillai, Brian A Yuan, James C Davis, and Aravind Machiry. An empirical study on the use of static analysis tools in open source embedded software. *arXiv preprint arXiv:2310.00205*, 2023.

[131] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, 2016.

[132] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295. IEEE, 2019.

[133] UAPI Group Specifications. Unified kernel image (uki). https://uapi-group.org/specifications/specs/unified_kernel_image/, 2025.

[134] Chad Spensky, Aravind Machiry, Nathan Burow, Hamed Okhravi, Rick Housley, Zhongshu Gu, Hani Jamjoom, Christopher Kruegel, and Giovanni Vigna. Glitching demystified: Analyzing control-flow-based glitching attacks and defenses. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 400–412, 2021.

[135] Christian Spinnler, Torsten Labs, and Norman Franchi. SoK: A taxonomy for hardware-based fingerprinting in the internet of things. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, pages 1–12. ACM.

[136] Nicholas Starke. U-boot fuzzing. https://starkeblog.com/qemu/u-boot/bootloader/fuzzing/negative-result/2021/03/12/u-boot-fuzzing.html, 2021.

[137] Nicholas Starke. Bootfuzz: A mbr fuzzer. https://github.com/nstarke/bootfuzz, 2024.

[138] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 32–41, New York, NY, USA, 1996. Association for Computing Machinery.

[139] Vincent Zimmer Subrata Banik. *System Firmware: An Essential Guide to Open Source and Embedded Solutions*. Apress, 2022.

[140] Priyanka Prakash Surve, Oleg Brodt, Mark Yampolskiy, Yuval Elovici, and Asaf Shabtai. SoK: Security below the OS – a security analysis of UEFI.

[141] SUSE. Cve-2024-49504 detail. https://nvd.nist.gov/vuln/detail/CVE-2024-49504, 2024.

[142] systemd. systemd: System and service manager. https://github.com/systemd/systemd, 2003.

[143] Binarly Research Team. Finding logofail: The dangers of image parsing during system boot. https://binarly.io/posts/finding_logofail_the_dangers_of_image_parsing_during_system_boot/, 2023.

[144] Binarly REsearch Team. The dark side of uefi: A technical deep-dive into cross-silicon exploitation. https://www.binarly.io/blog/the-dark-side-of-uefi-a-technical-deep-dive-into -cross-silicon-exploitation, 2024.

[145] Red Hat Bootloader Team. shim, a first-stage uefi bootloader. https://github.com/rhboot/shim, 2012.

[146] Runtime Reconfigure Team. How to test and validate firmware in hardware-in-the-loop (hil) environments, 2024.

[147] Runtime Reconfigure Team. Bootloader development 101: Best practices and common pitfalls, 2025.

[148] Tianocore. Edk-2. https://github.com/tianocore/edk2, 2007.

[149] Tianocore. A tour beyond bios - security enhancement to mitigate buffer overflow in uefi. https://edk2-docs.gitbook.io/a-tour-beyond-bios-mitigate-buffer-overflow-in-ue , 2020.

[150] tpm2dev. Boot with tpm: Secure vs verified vs measured, 2021.

[151] Trellix. Technique: Glitching u-boot (or other bootloaders) by shorting the nand flash. https://www.trellix.com/assets/docs/atr-library/ms-glitching-uboot_als5.pdf, 2022.

[152] U-Boot. Measured boot. https://docs.u-boot.org/en/latest/usage/measured_boot.html.

[153] u boot. u-boot. https://github.com/u-boot/u-boot.

[154] Jayakrishna Vadayath, Moritz Eckert, Kyle Zeng, Nicolaas Weideman, Gokulkrishna Praveen Menon, Yanick Fratantonio, Davide Balzarotti, Adam Doupé, Ruoyu Wang, Christophe Hauser, and Yan Shoshitaishvili. Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs. In *Proceedings of the 31st USENIX Security Symposium*, 2022.

[155] Stephan Van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Daniel Genkin, Andrew Miller, Eyal Ronen, Yuval Yarom, and Christina Garman. SoK: SGX.fail: How stuff gets eXposed. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4143–4162. IEEE.

[156] Jianqiang Wang, Meng Wang, Qinying Wang, Nils Langius, Li Shi, Ali Abbasi, and Thorsten Holz. A Comprehensive Memory Safety Analysis of Bootloaders. In *Proceedings of the 2025 IEEE Symposium on Security and Privacy*, 2025.

[157] Windows. Boot and uefi. https://learn.microsoft.com/en-us/windows-hardware/drivers/bringup/boot-and-uefi.

[158] Windows. Overview of boot options in windows. https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/boot-options-in-windows.

[159] Windows. Secure the windows boot process. https://learn.microsoft.com/en-us/windows/security/operating-system-security/system-security/secure-the-windows-10-boot-process.

[160] Windows. Trusted platform module technology overview. https://learn.microsoft.com/en-us/windows/security/hardware-security/tpm/trusted-platform-module-overview.

[161] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 304–316, New York, NY, USA, 2002. Association for Computing Machinery.

[162] wolfSSL. Top 10 things you should know about secure boot, 2023.

[163] Christopher Wright, William A. Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A. Clements. Challenges in firmware rehosting, emulation, and analysis. *ACM Comput. Surv.*, 54(1), January 2021.

[164] Fanxiao Xing, Lukai Li, Xiao Li, and Qichao Yang. Sftaint: Leveraging precise taint source recognition to enhance efficiency of static taint analysis in embedded systems. In *Proceedings of the 3rd International Conference on Signal Processing, Computer Networks and Communications*, SPCNC '24, page 272–279, New York, NY, USA, 2025. Association for Computing Machinery.

[165] Huaishuo Yan and Baojiang Cui. Uefuzzer: Enabling struct-aware fuzzing on uefi with static analysis. In *Proceedings of the 2025 5th International Conference on Computer Network Security and Software Engineering*, CNSSE '25, page 350–355, New York, NY, USA, 2025. Association for Computing Machinery.

[166] Jingyu Yang, Guize Liu, Jinsong Ma, and Weikun Yang. UbootKit: A Worm Attack for the Bootloader of IoT Devices. In *Virus Bulletin Conference*, 2019.

[167] Z. Yang, Y. Viktorov, J. Yang, J. Yao, and V. Zimmer. Uefi firmware fuzzing with simics virtual platform. *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[168] Jiewen Yao. Enable aslr for uefi in edk ii. https://edk2-docs.gitbook.io/a-tour-beyond-bios-mitigate-buffer-overflow-in-ue/address_space_layout_randomization/enable_aslr_for_uefi_in_edkii, 2020.

[169] J. Yin, X. Wang, D. Wei, J. Chen, Y. Zhou, Z. Li, and Y. Liu. Finding smm privilege-escalation vulnerabilities in uefi firmware with protocol-centric static analysis. *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1623–1637, 2022.

[170] Jiawei Yin, Menghao Li, Yuekang Li, Yong Yu, Boru Lin, Yanyan Zou, Yang Liu, Wei Huo, and Jingling Xue. RSFuzzer: Discovering Deep SMI Handler Vulnerabilities in UEFI Firmware with Hybrid Fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2155–2169, May 2023. ISSN: 2375-1207.

[171] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proceedings 2014 Network and Distributed System Security Symposium*, San Diego, CA, 2014. Internet Society.

[172] Kyle Zeng. top4grep: A grep tool for the top 4 security conferences. https://github.com/Kyle-Kyle/top4grep, 2023.

[173] Fengwei Zhang and Hongwei Zhang. SoK: A study of using hardware-assisted isolated execution environments for security. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pages 1–8. ACM.

[174] Yuan Zhang, Lei Qu, Yifei Wu, Leihuan Wu, Tingting Yu, Rui Chen, and Weiqiang Kong. Bounded verification of atomicity violations for interrupt-driven programs via lazy sequentialization. *ACM Trans. Softw. Eng. Methodol.*, 34(3), February 2025.

[175] Jiaxu Zhao, Yuekang Li, Yanyan Zou, Zhaohui Liang, Yang Xiao, Yeting Li, Bingwei Peng, Nanyu Zhong, Xinyi Wang, Wei Wang, et al. Leveraging semantic relations in code and data to enhance taint analysis of embedded systems. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7067–7084, 2024.

[176] Lianying Zhao, He Shuang, Shengjie Xu, Wei Huang, Rongzhen Cui, Pushkar Bettadpur, and David Lie. SoK: Hardware security support for trustworthy execution.

[177] Jie Zhou, John Criswell, and Michael Hicks. Fat pointers for temporal memory safety of c. *Proc. ACM Program. Lang.*, 7(OOPSLA1), April 2023.

Appendix A.
Notations

We use a two-dimensional notation combining **symbols** and **colors**. ✔ and ✖ indicate presence or absence; green marks desirable features and red marks undesirable ones. Thus ✔ means present and desirable, ✔ means present but undesirable, and ⚠ denotes partial support.

Appendix B.
Tool Configuration

Table 7: Bootloaders and Corresponding Known Vulnerabilities.

Bootloader	Commit Hash	# Known Vulns	Associated CVEs
EDK-II	0395045ae3	16	CVE-2022-36765, CVE-2023-45229, CVE-2023-45230 CVE-2023-45231, CVE-2023-45232, CVE-2023-45233 CVE-2023-45234, CVE-2023-45235, CVE-2023-45236 CVE-2023-45237, CVE-2024-6119, CVE-2024-25742
coreboot	092a1398f6	8	N/A
SeaBios	b9b923ed45	3	N/A
GRUB2	f1ce0e15e7	29	CVE-2021-3695, CVE-2021-3697, CVE-2021-3981 CVE-2022-2601, CVE-2022-28733, CVE-2022-28734 CVE-2022-28735, CVE-2022-28736, CVE-2022-3775 CVE-2023-4692, CVE-2023-4693
shim	f27182695d	11	CVE-2023-40546, CVE-2023-40547, CVE-2023-40548 CVE-2023-40549, CVE-2023-40550, CVE-2023-40551 CVE-2023-4692, CVE-2023-4693, CVE-2024-2312
U-Boot	89ab1e2817	12	CVE-2016-9841, CVE-2018-25032, CVE-2022-2347 CVE-2022-30552, CVE-2022-30790, CVE-2022-37434 CVE-2023-43804, CVE-2023-45803, CVE-2023-52323 CVE-2024-22195, CVE-2024-3651
TF-A	06796a08d3	11	CVE-2017-5715, CVE-2022-23960, CVE-2022-47630 CVE-2023-49100

We evaluated the tools in Table 7: 7 were closed-source and 4 required specialized hardware, so both groups were excluded. Below we describe the configuration for the remaining tools.

STASE. No special configuration; we followed the repository README.

FwHunt. We used the provided Docker container to test compiled EDK-II binaries and cross-referenced results against vendor binaries.

BootStomp. We ported the IDA Pro taint-tracing scripts to a newer IDA version; the updated scripts limited new test generation, but existing seeds reproduced the documented results.

EMBA. Uses FwHunt internally, so excluded to avoid duplication.

FuzzUEr. A minor source patch was required; interfaces were scoped to those with known reachable bugs to avoid unnecessary overhead.

UEFIscanner. A Windows runtime enforcement tool; not testable against custom EDK-II builds.

HBFA. Run inside a Docker image following the documentation; all harnesses were executed against the evaluation version of EDK-II.

efi_fuzz. Targets NVRAM variables only; configuration consisted of compiling EDK-II to produce the required binary.

BootFuzz. Minimal configuration: compile SeaBIOS and point the fuzzer to the custom build.

Emmutaler. Requires Apple bootloader binaries extracted from a physical device; no repository binaries were provided.

ASPFuzz. Provides a modified QEMU; excluded because sufficient ground-truth data was unavailable.

arbiter. Crashed on all three bootloader types; fixing it is outside the scope of this SoK.

CodeQL. No special configuration; we ensured the build included the drivers containing injected vulnerabilities.

Angr. Requires custom scripts per codebase; creating a harness is outside scope.

TSFFS. Integrated into FuzzUEr and tested as part of that evaluation.

Appendix C.
Exploit Mitigation

Exploit mitigation techniques do not prevent vulnerabilities themselves but significantly reduce the likelihood of weaponizing them, *e.g.,* turning a buffer overflow into code execution. We focus on common exploit mitigation techniques, specifically, ASLR [43], Stack Canaries [30], NX memory [14], PAC [97], CFI [2], and RELRO [29].

Table 8: Enabled Exploit Mitigation Prevalence in Bootloaders.

Bootloaders	ASLR	Stack Canaries	NX	PAC	CFI	RELRO
Open-Source (39)	0	0	7	1	0	1
Closed-Source (4)	0	0	2	0	0	0

C.1. Prevalence Study

For each bootloader we collected, we determined whether these mitigations are enabled. Open-source bootloaders were analyzed via manual documentation and source-code review, while closed-source bootloaders were evaluated with `checksec` [73]. Table 8 shows the summary. Overall, most bootloaders lack exploit mitigations due to structural and semantic constraints. We discuss each mitigation, its challenges in bootloader contexts, and its prevalence.

C.2. Address Space Layout Randomization

This technique randomizes the memory layout of code, stack, heap, and other sections to introduce non-determinism in memory addresses to hinder return-oriented programming (ROP) and jump-oriented programming (JOP) exploits [43]. As shown in Table 8, none of the bootloaders have ASLR enabled. Two requirements are critical for this: (i) relocatable sections/regions (position-independent code and data) and (ii) a reliable entropy source for unpredictable offsets [52]. While straightforward in userspace or OS environments, bootloaders face additional constraints.

First, many bootloaders rely on a fixed memory layout for deterministic hardware initialization. Low-level routines and memory-mapped structures often require absolute addresses, making position independence difficult. OS-level ASLR typically leverages virtual memory, but early boot stages lack an MMU, forcing relocations in physical memory, which risks overwriting critical regions. Trusted Firmware-A and EDK-II enable the MMU in stage 2, allowing randomized mappings after initialization [6], [47]. In contrast, U-Boot's SPL executes without an MMU, copying itself and driver blobs into fixed SRAM addresses [32].

Second, entropy sources such as hardware based random number generators (TRNGs), timing jitter, or environmental noise are required for unpredictability. Each bootloader stage must derive fresh entropy, since weak sources (*e.g.,* a timestamp counter before RNG initialization) can produce predictable layouts [168]. High-quality entropy can also impact memory usage and boot time, particularly when separate memory regions like TrustZone require distinct randomization [149].

C.3. Non-Executable Memory

This technique enforces separation of memory into executable and non-executable regions, preventing code execution from data sections such as the stack or heap [14]. As shown in Table 8, overall 9 bootloaders have NX enabled. Although simple, enabling NX has the following requirements.

First, hardware support (MMU or MPU) is required for per-page or per-region execute permissions. Early boot stages often map memory uniformly, making selective NX enforcement difficult (§ 2).

Second, proper NX requires initializing page tables or MPU regions before executing untrusted code, careful linker section layouts, and handling dynamic allocations that may need executable permissions. Some legacy routines rely on self-modifying code, creating unavoidable exceptions. Bootloader support varies, for example, TF-A and EDK-II enable NX after stage 2 [144], [149], while SeaBIOS cannot due to 16-bit real mode constraints.

C.4. Pointer Authentication Codes

PAC embeds cryptographic signatures in pointers to detect and prevent modification [97], protecting return addresses and function pointers from corruption.

First, PAC requires CPU support (*e.g.,* ARMv8.3) and a compiler that emits PAC instructions. Legacy hardware cannot leverage PAC without upgrades or software fallbacks [117].

Second, PAC keys must be securely stored and initialized early. Stages before key setup remain vulnerable, and incorrect initialization can halt execution [117].

Third, all protected pointers (*e.g.,* return addresses, function pointers) must be compiled or annotated with PAC instrumentation [117]. Exception handling and legacy code using pointer arithmetic or packed structures may require refactoring. For example, physical addresses mapped to peripherals may require careful PAC instrumentation. Despite these challenges, as shown in Table 8, it is interesting to see that one bootloader (*i.e.,* TF-A) has PAC enabled.

C.5. Stack Canaries

Stack canaries place a sentinel adjacent to control-flow data and verify it on function exit [30]. This mitigates classic stack-based attacks that overwrite return addresses or saved frame pointers. Despite their popularity, *none of the bootloaders have stack canaries enabled, as shown in Table 8.*

First, compiler support is required to insert and check canaries consistently. Hand-written assembly, leaf functions, and interrupt handlers may bypass instrumentation, leaving unprotected paths, often requiring manual auditing or reimplementation.

Second, a fresh random canary must be generated each boot using a high-entropy source. Early boot stages may lack quality entropy, producing predictable values and reducing effectiveness [144], [149].

Third, stack layout must accommodate canaries without disrupting local variables or alignment. Adding canaries increases stack usage and check instructions, potentially impacting performance in time-critical stages [144].

C.6. Control-Flow Integrity

CFI ensures execution follows legitimate control-flow paths, preventing hijacking of indirect branches [61]. Control Flow Guard (CFG) is a software implementation [115], and hardware-assisted enforcement, such as Intel CET, adds shadow stacks and forward-edge protection [66]. In bootloaders, these protect low-level initialization and prevent ROP/JOP attacks. As shown in Table 8, none of the bootloaders have CFI (or its variants) enabled, *e.g.,* forward edge protection.

First, compiler and linker support is required to generate metadata describing valid targets and preserve function/block identifiers [110]. Static linking, aggressive optimization, and hand-written assembly can break metadata consistency, causing false positives or bypasses. Self-modifying code and jump tables further complicate software CFI.

Second, runtime verification is needed for all indirect branches, either software-inserted or hardware-assisted [2]. Hardware enforcement reduces overhead but requires modern CPUs, limiting portability. Early boot stages often use minimal runtime and assembly, making consistent CFI integration challenging.

Third, bootloaders must coordinate control-flow design with initialization routines. Shadow stacks and branch tracking must be initialized correctly, and indirect branches must comply. Metadata/code mismatches can halt the boot process or trigger false positives [61].

C.7. Relocation Read-Only

RELRO marks relocation tables (*i.e.,* GOT, PLT) as read-only after dynamic relocations, preventing modification [29], [45]. As shown in Table 8, only one bootloader has RELRO enabled. It mitigates attacks targeting writable relocation entries to hijack control flow.

First, an MMU or equivalent is required to mark pages writable during relocation and read-only afterward. Early boot stages without MMU cannot enforce RELRO. Static linking with minimal relocation data limits applicability, and adding metadata increases image size and complexity [156].

Second, relocation entries must be applied before marking pages read-only. This requires linker scripts to separate writable relocation sections from executable code. Incorrect separation can cause legitimate accesses to fail, potentially leading to boot failures [86]. Late-stage modules needing relocations are incompatible with strict RELRO.

Third, dynamic relocation parsing must be integrated into the bootloader to benefit from RELRO, increasing memory usage and code complexity [86]. For example, in type 1 bootloaders the MMU isn't initialized until the end of stage 2 (Figure 1), so enforcing RELRO at that point would fail.

Appendix D. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

D.1. Summary

Despite bootloaders being a critical component of most computer systems, they have been poorly covered in both breadth and depth in the scientific literature. This SoK paper addresses this problem by presenting a study of modern bootloaders, consisting of a survey of 43 bootloader implementations. The main contribution of the paper is an evaluation of the effectiveness of 25 vulnerability detection tools across six threat surfaces identified in a representable sample of bootloaders in the survey. The evaluation highlights that many of these tools are ineffective against the majority of identified threats, as they focus only on a limited subset of vulnerabilities and bootloader categories.

D.2. Scientific Contributions

- Provides a New Data Set For Public Use.
- Creates a New Tool to Enable Future Science.
- Identifies an Impactful Vulnerability.
- Provides a Valuable Step Forward in an Established Field.
- Establishes a New Research Direction.

D.3. Reasons for Acceptance

1) The paper provides a new data set for public use. The paper presents a new vulnerability dataset, BOOTBENCH, consisting of 3,658 vulnerabilities in bootloader implementations. The authors have committed to open sourcing their dataset.
2) The paper provides a comprehensive survey of real-world, open-source bootloader implementations.
3) The paper contributes to identifying impactful vulnerabilities through a clear description of bootloader attack surfaces, and identifying previously underexplored surfaces.
4) The paper provides a valuable step forward in an established field. The reviewers found an SoK with a broad survey of bootloader security to be a valuable addition to the literature.
5) The paper establishes a new research direction by highlighting seven open problems in bootloader security without adequate solutions.

D.4. Noteworthy Concerns

None