




Article

Delving Deep into Reverse Engineering of UEFI Firmwares via Human Interface Infrastructure

Siyi Chen ¹, Yu-An Tan ², Kefan Qiu ², Zheng Zhang ², Yuanzhang Li ¹ and Quanxin Zhang ^{1,*}¹ School of Computer Science & Technology, Beijing Institute of Technology, Beijing 100081, China; chensiyi@bit.edu.cn (S.C.); popular@bit.edu.cn (Y.L.)² School of Cyberspace Science & Technology, Beijing Institute of Technology, Beijing 100081, China; tan2008@bit.edu.cn (Y.-A.T.); kfqiu@bit.edu.cn (K.Q.); zhangzheng@bit.edu.cn (Z.Z.)

* Correspondence: zhangqx@bit.edu.cn

Abstract: The Unified Extensible Firmware Interface (UEFI) provides a specification of the software interface between an OS and its underlying platform firmware. UEFI UI is an interactive interface that allows users to configure and manage UEFI settings, which is closely related to HII (Human Interface Infrastructure). In practice, HII provides a mechanism that allows developers to create UI elements with HII-related protocols. In this paper, we provide a comprehensive analysis of the UEFI combined with a case study. We proposed a protocol-centered static analysis method to obtain UEFI's password policy, using HII-related protocols to find password implementation. Existing static analyses are ineffective in detecting such password policy in stripped UEFI firmware images. By reverse-engineering the IFR (Internal Forms Representation) in HII, we located where much sensitive information is stored. Lastly, we studied hardware port configurations, using Secure Boot as a case in point. We analyzed how UEFI uses the HII protocol to set relevant information in the UEFI UI. This paper is the first to offer a reverse-engineering systematic analysis of exploring UEFI via HII, providing valuable insights into its structure and potential enhancements for firmware security.

Keywords: UEFI firmware; HII; UEFI UI; password policy; sensitive information; hardware configuration



Citation: Chen, S.; Tan, Y.-A.; Qiu, K.; Zhang, Z.; Li, Y.; Zhang, Q. Delving Deep into Reverse Engineering of UEFI Firmwares via Human Interface Infrastructure. *Electronics* **2023**, *12*, 4601. <https://doi.org/10.3390/electronics12224601>

Academic Editor: Wojciech Mazurczyk

Received: 9 October 2023

Revised: 6 November 2023

Accepted: 8 November 2023

Published: 10 November 2023



Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The Unified Extensible Firmware Interface (UEFI) [1] is currently the most popular industry standard, defining the software interface between an operating system and its underlying platform firmware. Many industry-leading technology companies have created the UEFI Forum, which defines the specifications of the interfaces [2] used by operating systems and Platform Initialization (PI) specifications. Now, manufacturers provide UEFI-compatible firmware images as a replacement for the Basic Input/Output System (BIOS). Compared to BIOS, UEFI has a better extensibility, a more flexible configuration, and a more standard boot process. The UEFI firmware [3], commonly named the UEFI BIOS on x86 platforms for historic reasons, is the main component of system firmware. The UEFI operating system runs on top of the UEFI firmware and can usually be viewed as a tandem of a UEFI application, serving as an operating system bootloader and a UEFI-compatible kernel, which is aware of the features brought by the UEFI firmware via the UEFI specification. Alongside UEFI, alternative open-source projects such as coreboot [4], libreboot [5], linuxboot [6], and u-boot [6] offer different approaches and philosophies to firmware initialization, focusing on modularity, security, and the freedom to customize the boot process to specific needs and hardware configurations.

HII (Human Interface Infrastructure) [7,8] is a part of the UEFI specification, which provides a standard framework for managing user interfaces based on UEFI systems. More specifically, HII provides a unified way to create and display user interfaces for BIOS settings, system configurations, boot options, etc. Beyond this, it also offers a mechanism to manage these user interfaces, including how to gather user input, how to change

system configurations, and how to display the results of the changes. By providing a standard interface and framework, it simplifies the work of hardware manufacturers and operating system developers in creating user interfaces within the UEFI environment, thereby improving development efficiency and reducing development costs. In order to provide more possibilities in system booting, hardware configuration, and system-level management, the UEFI Management System User Interface (UEFI UI) based on HII came into being. This is an interactive interface that allows users to configure and manage UEFI settings. This user interface can be provided by manufacturers and may vary depending on the manufacturer and device. Typically, users can access this user interface by pressing a specific key (such as F2, F10, or Delete) when the computer starts up. In the UEFI UI, users can adjust various hardware and boot options, such as modifying the boot order, enabling or disabling devices, enabling Secure Boot [9], etc. Therefore, it is a very important tool that allows users to customize and adjust their computer systems.

By delving into the boot process of UEFI, we find that the UEFI UI is intimately connected with the BDS (Boot Device Select) phase. For example, changing the boot device sequence will impact the order in which boot devices are selected during the BDS phase. During the BDS phase, the system introduces the HII (Human Interface Infrastructure) framework, which facilitates the design of elegant human–computer interaction interfaces and even the extension of the functionality of the UEFI UI by maintenance personnel. This enables UEFI to offer a wealth of features compared to the traditional BIOS. These features could potentially lead to unauthorized access. Moreover, the complexity of the UEFI UI might lead to configuration errors, thereby affecting system security. Therefore, this paper focuses on a reverse-engineering exploration of the implementation details of the UEFI UI, conducts an in-depth, comprehensive analysis of it, and ultimately aims to enhance manufacturers' awareness of the security aspects of the UEFI.

However, the complexity and proprietary nature of UEFI binary images make reverse engineering challenging. To address this issue, researchers have used many specialized tools, such as the UEFI Tool [10] and IDA Pro [11]. These tools can help researchers understand and analyze the structure and content of UEFI binary images. However, these tools alone are far from sufficient for analyzing the user interface. To achieve this goal, this paper will use reverse engineering tools, like the UEFI Tool and IDA Pro, and delve deeply into the crucial role that the HII framework plays in the analysis of the UEFI UI. By applying existing reverse engineering techniques to the analysis of the UEFI UI, we reveal key information hidden in the firmware as well as some design flaws.

In this paper, we systematically investigate the password policy of UEFI. For systems that have set a UEFI password, we need to enter the correct password to gain access when we power on the system or enter the UEFI UI. Therefore, the UEFI password often becomes our first interaction with the user interface, sparking our intense interest to deeply analyze the implementation of the password policy. However, in many UEFI firmware, manufacturers have implemented various protection measures such as code obfuscation techniques, making it challenging for us to find the password implementation program via direct reverse engineering. Hence, to address this situation, we propose a protocol-centric static analysis method to analyze the UEFI password policy.

In addition, we aim to find the actual location of the password within the flash chip. Therefore, by analyzing the HII framework and combining it with the open-source tool Universal-IFR-Extractor [12], we reverse-engineered the VFR language that implements the UEFI UI. As a result, we found the storage locations of the password and other variables. Moreover, during this process, by validating all the modules of the Dell r730 firmware against the manufacturer's standard firmware, we found that not only is the UEFI password stored within the firmware, but previously used passwords are also recorded. This is because NVRAM variables were stored in the UEFI firmware by developers, which undeniably presents a significant security risk.

Finally, regarding the richer configuration options provided by UEFI, we studied the hardware port configurations, using secure boot as a case study. Additionally, com-

binning the HII framework and IDA, we analyzed the logic of the UEFI UI in setting hardware interfaces.

In summary, we make the following contributions:

- We analyzed the UEFI password policy of different manufacturers, using HII protocols. For the first time, from a reverse engineering perspective, we found the way that the AMI standard protocol implements password policy. We proposed a protocol-centered static analysis method for password policies.
- We deeply analyzed the role of the VFR language in the UEFI UI from a reverse engineering perspective. we found the storage locations of some sensitive information in the UEFI firmware image. Ultimately, during the analysis, we discovered that the UEFI firmware used by a series of models exposes the encrypted password content and salt value in the binary firmware and even records previously used passwords and salt values.
- Compared to traditional BIOS, UEFI offers richer features. We analyzed how UEFI configures hardware ports by reverse engineering the VFR language.

The rest of this work is structured as follows: Section 2 provides a comprehensive background; Section 3 describes the HII framework and its related technologies; Section 4 provides implementation details of the UEFI management Interface analysis process; Section 5 describes related work; Section 6 discusses additional research questions; and Section 7 concludes.

2. Background

2.1. Visual Forms' Representation

Another innovation of UEFI over Legacy BIOS is in the specification of building blocks for user interfaces [8]. There are various scenarios where a platform component might want to interact in some fashion with the user. Examples of this are when presenting a user with several choices of information, sending information to the display, or offering a user menus for configuring the system, such as the BIOS Setup program shown in Figure 1. The potentially arduous task of managing user interface elements is simplified for UEFI modules requiring such functionality by UEFI providing some basic graphical elements like forms, strings, images, and fonts.

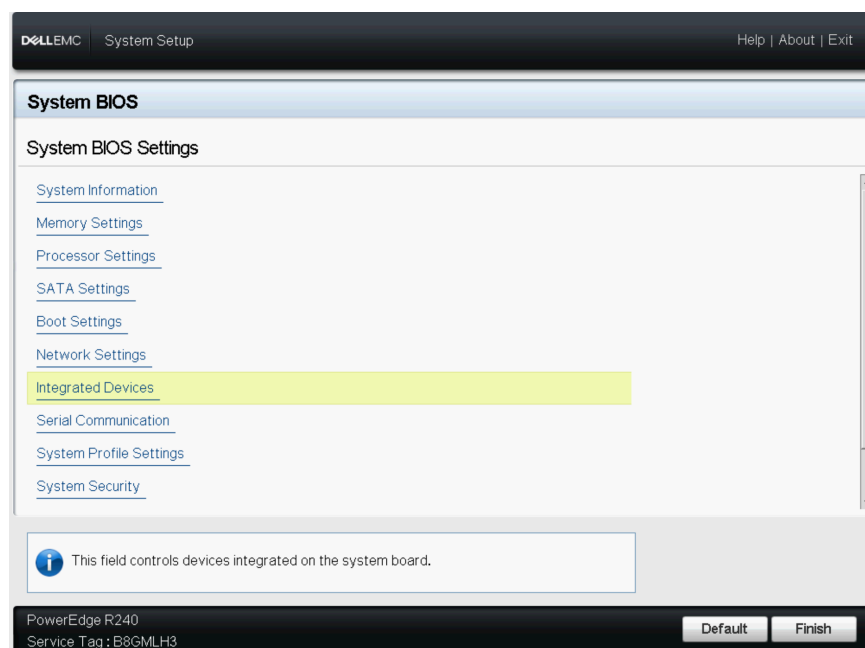


Figure 1. UEFI Management System User Interface.

UEFI calls this user interface support Human Interface Infrastructure (HII). HII is a set of protocols that allows for a UEFI module to register user interface and configuration content with the BIOS.

The implementation of the UEFI user interaction interface involves various types of files, such as VFR (Visual Forms Representation) and UNI. The strings of the UEFI interactive interface come from the UNI files, while the framework of the entire form is derived from the VFR files. In UEFI, there are roughly four components that constitute such a form: Strings, Forms, Fonts, and Images, as shown in Figure 2.

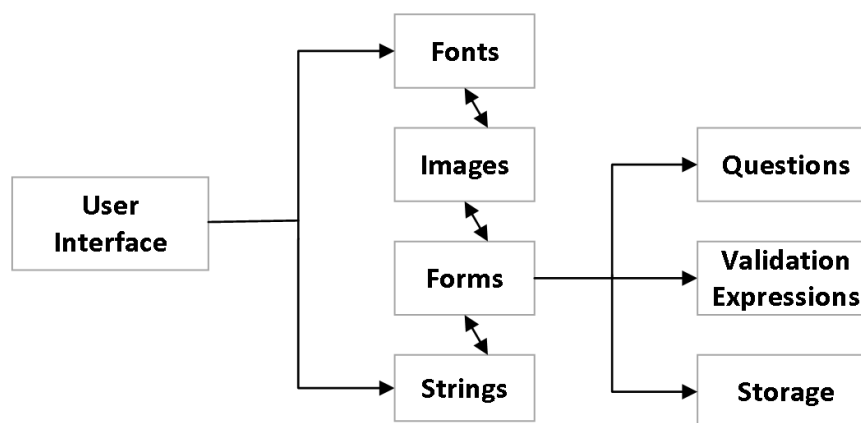


Figure 2. UEFI User Interface Components.

Strings are provided by UNI files, while Forms are supplied by VFR files. Forms describe the organization of the window, offer methods for user interaction, and dictate how the interactive content is stored. It is provided in a binary format, which, within the EDK framework, is referred to as IFR (Internal Forms Representation). IFR is generated by compiling VFR. Therefore, we define the organizational structure, interaction methods, and related content of the UEFI interactive interface by writing VFR files.

2.2. Static Code Analysis

Static Code Analysis refers to a code analysis technique where the program code is scanned using lexical analysis, syntax analysis, control flow analysis, data flow analysis, and other technologies without executing the code. This is performed to verify whether the code meets specifications, security, reliability, maintainability, and other metrics.

Programmers often make minor mistakes, such as missing a semicolon here or adding an extra parenthesis there. Most of the time, these oversights do not result in serious consequences: the compiler flags these errors, programmers correct the code, and the development process continues. However, this rapid feedback and response loop typically does not apply to most security vulnerabilities, which might lurk undetected for extended periods. As mentioned earlier, the longer software defects go unnoticed, the higher the cost to remedy them. The objective of static analysis is to inspect a program's text without attempting to execute it. Theoretically, static analysis tools can examine either the program's source code or its compiled form, with both offering similar advantages, although decoding the latter might be somewhat challenging [13].

Manual review or auditing is a form of static analysis and is highly time-consuming. For it to be effectively executed, human code auditors first need to know which types of mistakes they should be looking for, before rigorously examining the code. Reviewing the application code can be performed at any stage of software development, but it is most effective when undergone early, as detecting and rectifying security vulnerabilities and quality deficiencies might be high in cost and risk later in the software development process. When these errors slip into the market and are discovered by customers, they can impact profits and tarnish a company's reputation. Reviews encompass not just the code, but all documentation, requirements, and designs generated by developers, as every step

in software development can harbor mistakes. Essentially, human-performed static code analysis can be categorized into two main types: self-review and third-party review, both of which are closely related to individual software processes and team software processes [14].

Static analysis tools have advantages over manual reviews because they operate faster, meaning they can assess programs more frequently. They encapsulate some of the knowledge required to perform such code analysis in a way that the tool operator does not need the same expertise level as a human auditor. Just as programmers can rely on compilers to consistently handle the nuances of language syntax, operators of good static analysis tools can effectively use the tool without delving into the finer details of more elusive errors.

2.3. Password Policy

In today's era, the ubiquity of passwords is an indisputable fact. Authentication is often executed via a combination of usernames and passwords [15]. As a result, passwords often serve as the sole barrier between potential attackers and the victim's information. Knowing someone's password allows an attacker to impersonate that individual in an online environment or access sensitive data intended only for that person. As society's reliance on password security deepens, it also becomes increasingly vulnerable to threats of password breaches.

Previous studies have shown that when given a choice, users tend to create simple passwords [16,17]. However, simple passwords are particularly easy for attackers [15–17]. As a result, many groups and organizations implement password policies, imposing restrictions on the passwords users can create and how they use them. A well-crafted policy might enhance the security of an organization [15,17–19].

Much of a password policy typically relates to the creation of passwords. For instance, a password creation policy might require passwords to be at least six characters long and contain at least one numerical character. However, there are several other aspects in the lifecycle of a password where password policies are relevant.

When designing password policies, it is crucial to consider human factors in addition to technical ones. While password policies might specify the encryption methods to be used on passwords, overly complex passwords might end up being written down by users because they cannot remember them [17]. Likewise, many password policies outline with whom users can share passwords and under what circumstances they should contact the administrator.

Today's formulation of password policies seems to be based more on experience and heuristics than on science or computation. While many organizations use passwords for security, there is not a universally accepted unified framework under which all these password policies can be understood and compared. Such a unified framework would promote better creation and understanding of password policies.

2.4. NVRAM

Modern memory hardware is either fast but volatile (DRAM) or persistent but slow (SSD/HDD). As a result, to prevent data loss, applications expend significant effort to persist runtime information onto non-volatile storage mediums. In this process, data structures in main memory often undergo costly transformations to reflect the distinct access characteristics of block storage devices and better utilize limited bandwidth. Non-Volatile Random Access Memory (NVRAM) is the latest advancement in memory technology, representing a unique opportunity to simplify the persistence we know. Unlike DRAM, NVRAM cells do not require periodic refreshes to maintain their state, allowing them to retain content even when power is lost. Performance analysis currently places NVRAM between DRAM and NAND-flash, but with ongoing research and development, its performance is expected to eventually meet or even surpass DRAM. By allocating data structures on NVRAM instead of DRAM, the application state can ultimately be persisted in-place without sacrificing speed or increasing software complexity [20].

UEFI provides booting and system management, relying on data stored in UEFI NVRAM. The UEFI environment itself may be susceptible to malware attacks. Moreover, the existence of UEFI NVRAM introduces certain security concerns, such as UEFI NVRAM often being bypassed by security software and computer forensics frequently overlooking the vast data stored in UEFI NVRAM [21]. Attackers could exploit the UEFI NVRAM storage space to stash illicitly obtained data (like credit card details) on the victim's/attacker's computer between the communication lines, without any traditional high-capacity storage access. Another example is data exfiltration in malware scenarios, which can be used to store encryption keys in use, or even for leaking personal, private data, leading to privacy issues. The data would only appear as highly volatile data in memory and network data streams, making it hard to capture for IT forensic purposes unless strategically prepared in advance. Vulnerabilities in NVRAM also pose threats to UEFI itself. Given the rich configuration capabilities of UEFI, attackers might attempt to exploit any vulnerabilities in NVRAM to alter configurations [22]. Unauthorized configuration changes could expose the system to additional risks or allow attackers to bypass security controls.

2.5. Challenges in Firmware Analysis

Low-power, single-purpose embedded devices (such as routers and IoT devices) are ubiquitous. While they have automated and simplified many aspects of users' lives, recent large-scale attacks have demonstrated that they pose a serious threat to the internet infrastructure due to their vast numbers. Unfortunately, the software on these systems relies on hardware and typically runs in unique, minimized environments with non-standard configurations, making security analysis particularly challenging. In the field of reverse engineering, the correct image base of firmware has very important significance for the reverse engineers to understand the firmware by building accurate cross references. However, for a large number of embedded system firmwares, the format is nonstandard and the image base is unknown. Thus, [23] presents a two-step method to determine the image base of firmwares for ARM-based devices. First, based on the storage characteristic of string in the firmware files and the encoding feature of literal pools that contain string addresses, we propose an algorithm called FIND-LP to recognize all possible literal pools in firmware. Second, we propose an algorithm called Determining image Base by Matching Literal Pools (DBMLP) to determine the image base. Addressing the non-standardized firmware formats, Ref. [24] describes a methodology for automatically determining the image base of firmware of ARM-based industrial control systems. Two algorithms, FIND-String and FIND-LDR, are presented that obtain the offsets of strings in firmware and the string addresses loaded by LDR instructions, respectively. Ref. [25] introduced Karonte, a static analysis approach capable of analyzing embedded device firmware by modeling and tracking multi-binary interactions. This method propagates tainted information between binary files to detect insecure interactions and identify vulnerabilities. Ref. [26] designed and implemented a new tool named FIE, which builds on the KLEE symbolic execution engine to provide a scalable platform for detecting errors in firmware programs of the popular MSP430 series microcontrollers. FIE incorporates new symbolic execution techniques, enabling it to verify the security properties of simple firmware common in practice. Ref. [27] designed a software framework that continuously executes given firmware binary files while bootstrapping inputs from off-the-shelf fuzzers, achieving hardware-independent and scalable firmware testing. This framework employs a novel technique called P2IM, which abstracts various peripherals and processes firmware I/O in real-time based on automatically generated models. P2IM is agnostic to peripheral designs and generic to firmware implementations, making it applicable to a wide range of embedded devices. Ref. [28] introduced FIRMSCOPE, an innovative static analysis system that analyzes Android firmware to expose unwanted functionalities in pre-installed applications using efficient and practical context-sensitive, flow-sensitive, field-sensitive, and partially object-sensitive taint analysis.

For UEFI, existing static analyses fail to effectively detect such vulnerabilities in stripped COTS UEFI firmware images, which are developed based on a custom callback mechanism that organizes callable functions into protocols identified by GUIDs. Leveraging this callback-based programming paradigm, Ref. [29] introduced the first static detection framework, SPENDER, built upon a novel protocol-centric analysis for efficiently and accurately discovering potential SMM privilege escalation vulnerabilities in UEFI firmware. The calling conventions in UEFI are so unusual that existing reverse engineering tools cannot handle them natively. Therefore, Ref. [30] developed an extension for Ghidra [31] to better analyze UEFI modules.

3. Understanding HII Framework and Related Technologies

3.1. HII Overview

HII is a set of protocols that allow a UEFI driver to provide the ability to register user interface and configuration content with the platform firmware. Unlike legacy option ROMs, the configuration of drivers and controllers is delayed until a platform management utility chooses to use the services of these protocols. UEFI drivers are not allowed to perform setup-like operations outside the context of these protocols. This means that a driver is not allowed to interact with the user outside the context of this protocol.

Figure 3 shows a basic platform configuration or “setup” model. The drivers and applications install elements (such as fonts, strings, images, and forms) into the HII Database, which acts as a central repository for the entire platform. The Forms Browser uses these elements to render the user interface on the display devices and receive information from the user via HID devices. When complete, the changes made by the user in the Forms Browser are saved, either to the UEFI global variable storage—(GetVariable() and SetVariable())—or to the variable storage provided by the individual drivers.

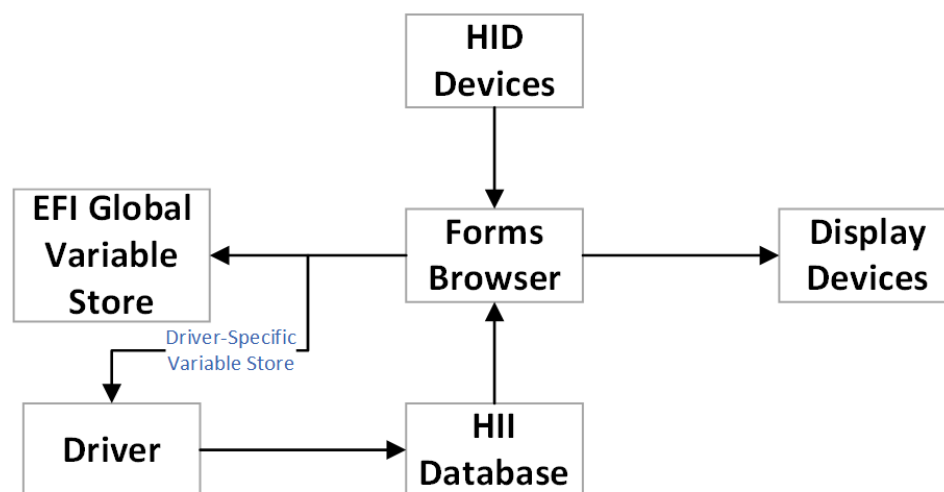


Figure 3. UEFI Platform Configuration Overview.

3.2. HII Elements

3.2.1. HII Databases

The HII database is the resource that serves as the repository of all the form, string, image, and font data for the system. Drivers that contain information destined for the end user will store their data in the HII database. For example, one UEFI module might implement the BIOS Setup program, allowing the user to configure motherboard component settings. Additionally, add-in cards may contain their own UEFI drivers, which, in turn, have their own BIOS Setup-related data. All the UEFI modules that contain BIOS Setup-related data can include their information in the HII database. This architecture is summarized in Figure 4.

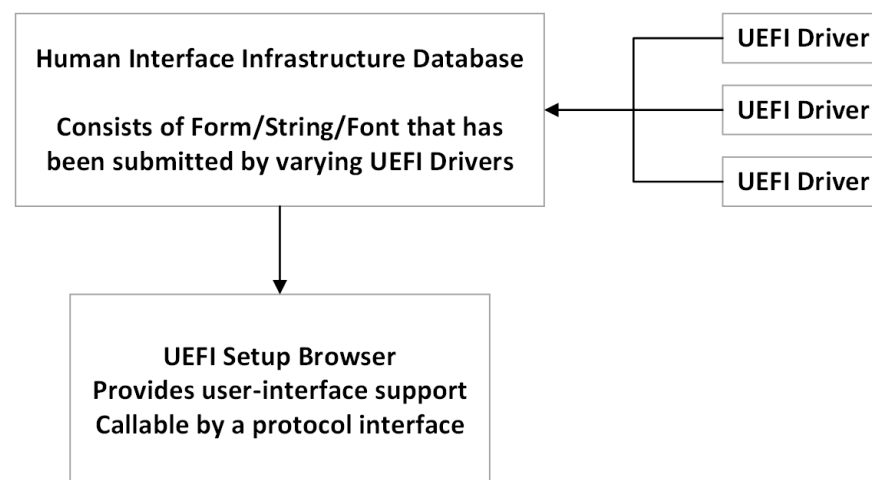


Figure 4. HII architecture.

3.2.2. Forms

The UEFI specification describes how a UEFI module can present a forms-based interface to a user and a UI element akin to Windows' CFrameWnd or Java's JFrame. This forms-based interface assumes that each window or screen consists of some window overhead, such as a title and buttons, and a list of user interface controls. For example, these controls could represent individual configuration settings for a UEFI application or driver. Associated with the notion of a form is a Forms Browser—the entity that reads the form data and presents a graphical representation on the display. The Forms Browser provides a forms-based renderer which understands how to read the contents of the forms, interact with the user, and save any resulting values. The Forms Browser uses forms data installed by UEFI modules in the HII database. The Forms Browser organizes the forms so that a user can navigate between the forms, select individual controls, and change the values using a keyboard, touch digitizer, or mouse. When the user has finished making modifications, the Forms Browser saves the values to NVRAM.

3.2.3. Strings

Strings in the UEFI environment are defined using the 16-bit UCS-2 character encoding. Strings are another one of the types of resources installed into the HII Database. In order to facilitate localization, programmers reference each string via a unique identifier defined as part of the strings package installed by the UEFI image. Each identifier may have several translations associated with it, e.g., English, French, and Traditional Chinese. When displaying a string, the Forms Browser selects the text to display based on the current platform language setting. The actual text for each language is stored in a separate file, which makes it possible to add and remove language support just by including or excluding language-specific files. Moreover, each string may have font information, including the font family name, font size, and font style, associated with it.

3.2.4. Images

UEFI supports storing images in the HII database. The format of images stored in the HII database was created to conform to the industry standard 1-bit, 4-bit, 8-bit, and 24-bit video memory layouts.

3.2.5. Fonts

UEFI specifies a standard font which is required for all systems that support text display on bit-mapped output devices. The standard font, named "system", is a fixed pitch font where all characters are either narrow (8×19 pixels) or wide (16×19 pixels). UEFI also allows for the display of other fonts, both fixed-pitch and variable-pitch. Platform support for fonts beyond the system is optional.

3.3. HII Protocol

HII Protocol is a set of protocols and services used for constructing and managing user interfaces. Its primary purpose is to provide a standardized way to describe and manage user interfaces in the UEFI management interface, including everything from BIOS setup menus to full-featured graphical user interfaces.

Through the analysis of these protocols, we propose a method to analyze the UEFI password policy produced by AMI vendors. To ascertain the storage location of the password policy in UEFI, we first need to be aware of protocols related to keyboard input, such as the EFI SIMPLE TEXT INPUT PROTOCOL. Due to the customizability of UEFI, many manufacturers redefine the keyboard input protocol based on this protocol. For instance, the AMI firmware manufacturer defines a keyboard input protocol named AMI EFIKEYCODE PROTOCOL as follows:

```
1 struct AMI_EFIKEYCODE_PROTOCOL{
2   AMI_RESET_EX Reset;
3   AMI_READ_EFI_KEY ReadEfikey;
4   EFI_EVENT WaitForKeyEx;
5   EFI_SET_STATE SetState;
6   EFI_REGISTER_KEYSTROKE_NOTIFY RegisterKeyNotify;
7   EFI_UNREGISTER_KEYSTROKE_NOTIFY UnregisterKeyNotify;
8 }
```

Based on the GUID of AMI EFIKEYCODE PROTOCOL as follows:

```
1 #define AMI_EFIKEYCODE_PROTOCOL_GUID{
2   0x0ADFB62D, 0xFF74, 0x484C,
3   {0x89, 0x44, 0xF8, 0x5C, 0x4B, 0xEA, 0x87, 0xA8}
4 }
```

We can accurately pinpoint the location in AMI firmware where this protocol is invoked. Subsequently, by analyzing the context and considering the characteristics of the password input, we can identify the password input location. After that, based on the program's execution logic, we can extract the UEFI password policy as illustrated in Figure 5.

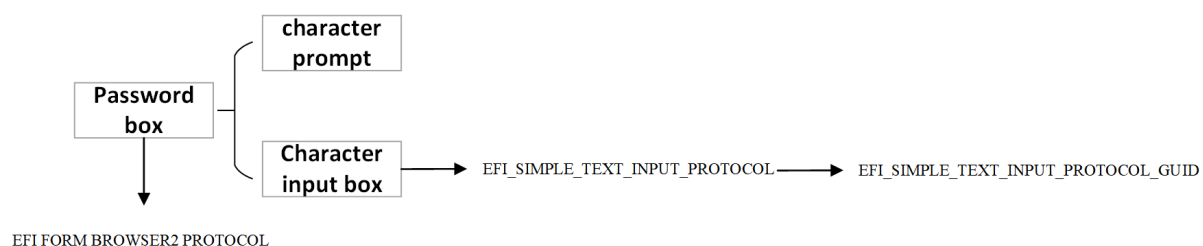


Figure 5. UEFI Password Input Box Model.

In HII, there is a protocol known as the HII Configuration Processing and Browser Protocol, which encompasses two other protocols: EFI FORM BROWSER2 PROTOCOL and EFI HII CONFIG ROUTING PROTOCOL. EFI FORM BROWSER2 PROTOCOL is used for displaying pages, and HII Drivers can retrieve information from current HII Driver configuration options that users have modified but not yet stored via its BrowserCallback function. EFI HII CONFIG ROUTING PROTOCOL is a global protocol that handles interactions on the Setup interface. We can implement some UEFI interface elements, such as drop-down boxes, based on these two protocols. Using the GUIDs of these two protocols, we can also locate their invocation points within the firmware as shown in Figure 6.

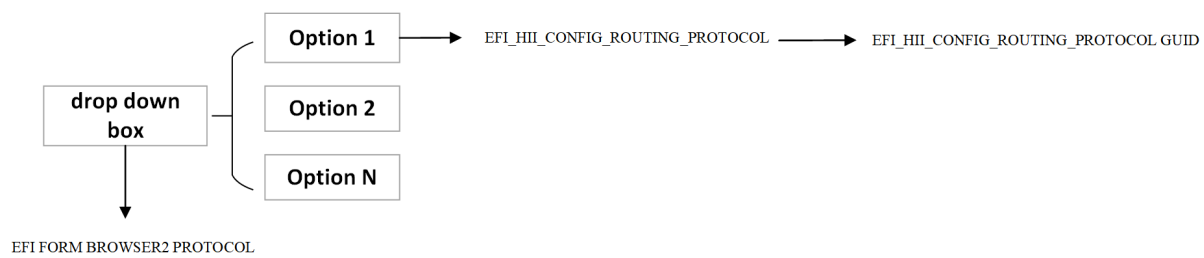


Figure 6. UEFI Dropdown Box Model.

3.4. Reverse Engineering of VFR

VFR (Visual Forms Representation) is a high-level descriptive language used to define the layout and behavior of the UEFI settings' interface. It is closer to source code, providing developers with a convenient way to design and describe forms, controls, and related actions. When VFR is compiled, it is transformed into IFR. IFR is the binary representation of VFR, stored in the UEFI firmware and parsed at runtime to display the corresponding user interface. Since it is in a binary format, directly reading or editing it is challenging; thus, we resort to VFR reverse engineering techniques to convert IFR back to VFR for easier comprehension. The reverse process from IFR to VFR can be employed to view or modify the UEFI firmware settings interface. The basic steps to transition from IFR to VFR are shown in Figure 7. Using tools like UEFITool, one can extract the IFR binary segment from the UEFI firmware image. As IFR is the compiled form of VFR, you need a decompiler to convert IFR back to VFR. Such tools might not be as common as forward compilers, but the community offers some, such as the Universal IFR Extractor. Once the VFR representation of the IFR is obtained, one can start analyzing it to understand the user interface layout, options, and behavior of the UEFI firmware.

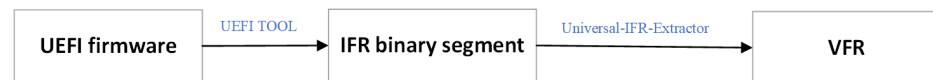


Figure 7. VFR Reverse Analysis Steps.

4. Case Studies

We have observed that the keyboard protocols associated with user input differ significantly among various vendors, so our approach to finding the password policy differs for firmware from different vendors. This paper initially uses Dell r240 as a case study to illustrate the simplest method for identifying the password policy. Then, taking Super-Server 6028U-TR4T+ as an example, this paper introduces a protocol-centric static analysis method to determine the password policy. Concerning the sensitive information of the UEFI UI, the primary focus of this paper is on the location of the password. By reverse analyzing VFR files, we pinpointed the location where the UEFI password is stored in NVRAM. Considering security aspects, the UEFI firmware has instituted protective measures for the location of the password. For instance, NVRAM variables are typically not stored within the UEFI firmware. However, this paper has discovered that for Dell r730 and r630 series models, the UEFI password is stored in the UEFI firmware file, verified via the integrity check of UEFI modules. Employing the rainbow table method, this paper has successfully retrieved the UEFI password. As for the implementation method of the hardware interface, referencing the aforementioned form implementation protocols, we can discern the program logic of the implemented interface by locating where these protocols are invoked in the firmware using IDA.

4.1. Password Policy Analysis

4.1.1. Analysis of Dell r240 UEFI Password Policy

In the Dell r240, we extracted the BdsDxe module, which is the driver that loads the UEFI UI and contains the program implementing the logic of the UI. We analyzed this

module via IDA and found strings related to the password, as shown in Figure 8. We then entered the function sub_2D0D0 that references this string for inspection as shown in Figure 9. We found there is a condition ' $v22 \geq 3$ '. Using the context information, we can understand that this ' $v22$ ' represents the number of password inputs. Therefore, we can determine that the number of inputs for this UEFI password cannot exceed three times; otherwise, the system will be locked.

```
aEnterSetupPass:                ; DATA XREF: sub_2D0D0+1B8f0
    text "UTF-16LE", 'Enter Setup Password: ',0
    align 10h
aPasswordInvalid:                ; DATA XREF: sub_2D0D0:loc_2D330f0
                                   ; sub_2D500:loc_2D808f0
    text "UTF-16LE", 0Ah
    text "UTF-16LE", '*** Password Invalid. ***',0Dh,0Ah,0
    align 10h
```

Figure 8. Password Strings in UEFI UI. We can find it by searching for strings in IDA.

```
if ( v22 >= 3 )
{
    v16 = L"\nNumber of unsuccessful password attempts: ";
    v23 = sub_3C9E0(L"\nNumber of unsuccessful password attempts: ");
    v26 = *(_QWORD *)(qword_75170 + 64);
    *(void (__fastcall *)(_QWORD, unsigned __int64, __int64))(v26 + 56)(
        *(_QWORD *)(qword_75170 + 64),
        40 - v23 / 2,
        16i64);
    *(void (__fastcall *)(_QWORD, const __int16 *))( *(_QWORD *)(qword_75170 + 64) + 8i64))(
        *(_QWORD *)(qword_75170 + 64),
        v16);
    sub_4B5F0(L" %d ", v22);
    v16 = L"\n*** Maximum number of password attempts exceeded. System halted. ***";
    v23 = sub_3C9E0(L"\n*** Maximum number of password attempts exceeded. System halted. ***");
    v27 = *(_QWORD *)(qword_75170 + 64);
    *(void (__fastcall *)(_QWORD, unsigned __int64, __int64))(v27 + 56)(
        *(_QWORD *)(qword_75170 + 64),
        40 - v23 / 2,
        17i64);
    *(void (__fastcall *)(_QWORD, const __int16 *))( *(_QWORD *)(qword_75170 + 64) + 8i64))(
        *(_QWORD *)(qword_75170 + 64),
        v16);
    sub_41660();
}
```

Figure 9. System Locked In Dell R240. The $v22$ contains the count of password entry attempts; when the number exceeds 3, the system gets locked.

4.1.2. Analysis of SuperServer 6028U-TR4T+ UEFI Password Policy

The UEFI firmware of SuperServer 6028U-TR4T+ is different from that of r240 in that there is no way to find the function that implements the UEFI password input via the same string search method. Therefore, this paper proposes a targeted method to solve this situation. The steps of this method are as follows:

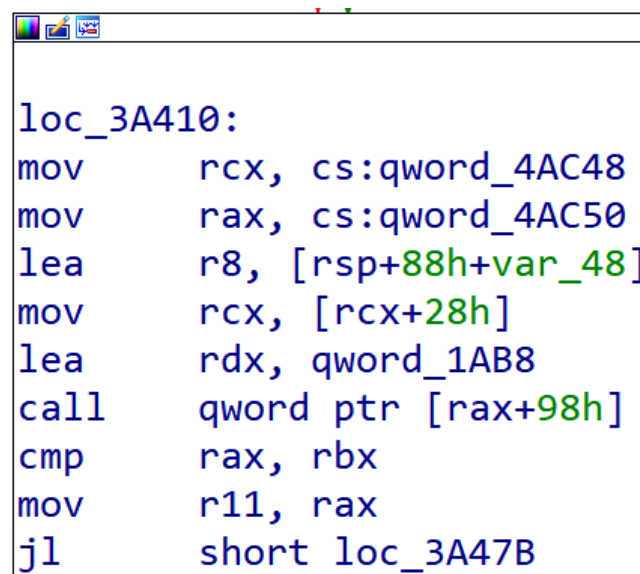
- Obtain the GUID of the protocol that implements the keycode input. In this paper, we obtained the AMI EFIKEYCODE PROTOCOL GUID 0x0ADFB62D, 0xFF74, 0x484C, {0x89, 0x44, 0xF8, 0x5C, 0x4B, 0xEA, 0x87, 0xA8}.
- Retrieve the module that implements the password logic from the UEFI image. In this paper, we obtained AMISetup, which in some firmware is known as BdsDxe.
- Considering the impact of endianness, we need to search for the GUID in the module using both big-endian and little-endian formats.
- After locating the GUID, we validate whether this protocol has been invoked here.
- After determining that there is an input operation on the interface, we check the context to confirm if it is the part where a password is being entered. For example, we can look to see if there is an operation that converts characters into asterisks (*).
- After confirming that it is the part where a password is entered, we iteratively examine the original function to observe the program's logic.

Through searching for the AMI EFIKEYCODE PROTOCOL GUID, we locate this line of code as shown in Figure 10.

```
while ( 1 )
{
    (*(void (__fastcall **)(_QWORD, __int64, __int64)))(*_QWORD *)byte_40 + 56i64))(*(_QWORD *)byte_40, v10, a4);
    LOBYTE(v11) = 1;
    (*(void (__fastcall **)(_QWORD, __int64)))(*_QWORD *)byte_40 + 64i64))(*(_QWORD *)byte_40, v11);
    do
    {
        v12 = (*(__int64 (__fastcall **)(_QWORD, __int64 *, __int64 *))&byte_40[88])(*_QWORD *)word_28, qword_1AB8, &v19);
        if ( v12 >= 0 )
        {
            v12 = (*(__int64 (__fastcall **)(_QWORD, __int64 *, __int64 *))&byte_40[88])(*_QWORD *)word_28, qword_1AB8, &v19);
            if ( (v21 & 0x40) != 0 && v12 >= 0 && !v20 )
            {
                v12 = 0x8000000000000000ui64;
                v17 = v20;
            }
        }
    }
}
```

Figure 10. System Calls The AMI EFIKEYCODE PROTOCOL. We located this spot by tracing the GUID.

We find that qword 1AB8 indeed matches 2D B6 DF 0A 74 FF 4C 48 89 44 F8 5C 4B EA 87 A8. So, we continue to confirm this point. By translating this code into assembly language as demonstrated in Figure 11, we identify a call instruction, of which we suspect this instruction calls the HandleProtocol function. Further analysis of this assembly instruction reveals an offset of 98 h. We searched the UEFI specification and discovered that the EFI BOOT SERVICES structure is defined within it, containing numerous function pointers. These function pointers are arranged in a specific order in the UEFI firmware. Hence, each function has an offset corresponding to its position in the structure. The offset of the HandleProtocol function in the EFI BOOT SERVICES structure under the X64 architecture is precisely 98 h. Therefore, this code is calling AMI EFIKEYCODE PROTOCOL.



```
loc_3A410:
mov     rcx, cs:qword_4AC48
mov     rax, cs:qword_4AC50
lea     r8, [rsp+88h+var_48]
mov     rcx, [rcx+28h]
lea     rdx, qword_1AB8
call    qword ptr [rax+98h]
cmp     rax, rbx
mov     r11, rax
j1      short loc_3A47B
```

Figure 11. Assembly Code Representation.

Then, we start looking for the context of this code and find some keyboard code information and the implementation of converting characters into asterisks, as shown in Figures 12 and 13.

```

if ( HIWORD(v17) == 8 )
{
    if ( v9 )
    {
        --v9;
        --v7;
        a1[v9] = 0;
    }
    goto LABEL_23;
}
if ( HIWORD(v17) == 13 )
{
    (**(void (__fastcall **)(_QWORD, _QWORD))&word_28[4])(*_QWORD *)&word_28[4], 0i64);
    sub_69B8(&v18);
    goto LABEL_32;
}

```

Figure 12. Keyboard Code Information. HIWORD(v17) == 8' means that when the keyboard code of the character entered by the user is equal to 8, then the backspace operation is executed. HIWORD(v17) == 13' means that when the keyboard code of the character entered by the user is equal to 13, the system performs a confirmation operation.

```

for ( i = 0i64; i < 0x14; ++i )
    *(_WORD *) (v14 + 2 * i) = i < v9 ? 42 : 32;

```

Figure 13. Converting The Password Input Chars With The Hidden Mark '*'.

We conclude that this part of the code is the part that implements the password logic and then we start looking at the original function that calls this part of the code, as shown in Figure 14.

```

while ( 1 )
{
    v12 = sub_3A300(v9, 20i64, v74, v73);
    if ( v12 != 0x8000000000000015ui64 )
    {
        if ( v12 == 0x8000000000000012ui64 )
            goto LABEL_23;
        v10 = v11 & (unsigned int)sub_39604(v9);
        if ( !*v9 )
        {
            if ( v10 )
                goto LABEL_23;
            if ( (v11 & 1) == 0 && (a1 & 1) != 0 )
                v10 = 1i64;
            if ( (v11 & 2) == 0 && (a1 & 2) != 0 )
                v10 = 2i64;
        }
        if ( v10 )
        {
            EL_23:
                v6 = v72[0];
                break;
        }
        if ( *a2 == 1i64 )
            sub_3A1B0(20, 19, v74, v73, 0);
        else
            sub_3A1B0(20, 19, v74, v73, 1);
        --*a2;
    }
    if ( !*a2 )
        goto LABEL_23;
}

```

Figure 14. 'sub 3A300' is the function where Figures 12 and 13 are located. Upon entering the original function, it was found that 'sub 3A300' is within a 'while' loop.

In the original function, sub 3A300 is nested inside a while loop, which ends when the value of *a2 is 0. We observe that *a2 is passed as an argument to int64 fastcall sub 39AA4(char a1, QWORD *a2), so we continue to jump to the original function that calls this while loop, as shown in Figure 15. We can see that v4 is passed to the parameter position

where *a2 is located. Since the value of v4 is 3, we can conclude that the maximum number of password attempts is 3. Finally, we find the number of password attempts using this protocol-centric static analysis method. All the AMI standard interface UEFI firmware we have tried can find the maximum number of password attempts using this method.

```
char sub_30F04()
{
    char v0; // bl
    unsigned int v1; // eax
    char v3; // [rsp+30h] [rbp+8h] BYREF
    __int64 v4; // [rsp+38h] [rbp+10h] BYREF
    char v5; // [rsp+40h] [rbp+18h] BYREF

    v0 = 0;
    byte_4AF7A = 1;
    v1 = sub_2EDE4();
    v4 = 3164;
    if ( (unsigned __int8)sub_2ED44(v1) )
    {
        v0 = 1;
        if ( !(unsigned int)sub_39AA4(0, &v4) )
        {
            while ( 1 )
            {
                (*(void (__fastcall *))(__int64, _QWORD, char *))&byte_40[32])(
                    1i64,
                    *(_QWORD *)(&word_28[4] + 16i64),
                    &v5);
                (*(void (__fastcall *)(_QWORD, char *)))(*_QWORD *)&word_28[4] + 8i64)(*_QWORD *)&word_28[4], &v3);
            }
        }
    }
    return v0;
}
```

Figure 15. v4 is passed as an argument into 'sub 39AA4', which is the original function of Figure 14. In Figure 14, a2 inherits the value of v4. With each loop, the value of a2 decreases by 1. Therefore, v4 stores the number of password attempts.

4.2. Sensitive Information Localization

Using SuperServer 6028U-TR4T+ as an example, the steps for locating sensitive information are as follows:

- Extract the corresponding module from the UEFI firmware using the UEFI TOOL; here, we are extracting the Setup module.
- Use the Universal-IFR-Extractor tool to decompile the Setup module into VFR language.
- From this VFR language information, we can retrieve the content of different sensitive information, taking the password as an example, as shown below.
- Find the corresponding NVRAM variable name and storage offset from these contents.
- Afterwards, we can view the NVRAM variables in the UEFI Shell for verification.

```
1 Gray Out If{19 82}
2 QuestionId: 0x224 equals value 0x1{12 06 24 02 01 00}
3 password: Administrator Password,
4     VarStoreInfo(VarOffset/VarName): 0x28,
5     VarStore: 0x6, QuestionId: 0x1B3,
6     MinSize: 0x3, MaxSize 0x14{08 91 CB 04 CC 04 B3 01 06 00 28 00 00 03 00
    ↪ 14 00}
7 End{29 02}
8 End If{29 02}
```

From the example above, we can see that in the NVRAM variable with VarStore 0x6, the administrator password is stored at an offset of 0x28. We further discover that the NVRAM variable with VarStore 0x6 is named AMITSESetup. However, a variable named AMITSESetup usually does not exist in UEFI firmware, as passwords are typically not stored within UEFI firmware. By verifying a computer that uses this UEFI firmware, we can locate the actual storage location of the AMITSESetup variable. After clearing the AMITSESetup variable, the administrator password is removed.

However, not all firmware will store the password in other locations on the chip. By verifying the Dell r730 firmware module file with the standard firmware module file provided by the manufacturer, we found that the Dell r730 firmware that has set the password will store password-related information in a padding file, including the



Figure 17. Selection Box.

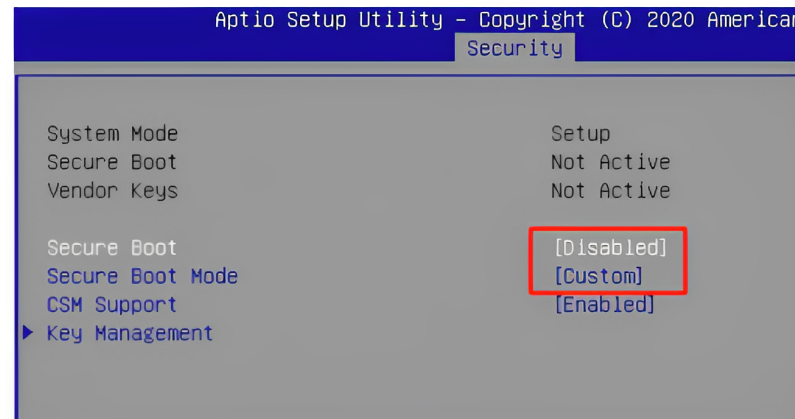


Figure 18. Dropdown Box.

In the previous section, we learned that implementing a drop-down box in UEFI can be achieved by using a combination of EFI FORM BROWSER2 PROTOCOL and EFI HII CONFIG ROUTING PROTOCOL. Therefore, in the decompiled module file, we expect to find the locations where these two protocols are referenced simultaneously. We first arrange the GUID of EFI FORM BROWSER2 PROTOCOL as 60 c3 d4 b9 fb bc 9b 4f 92 98 53 c1 36 98 22 58 as follows:

```
1 #define EFI_FORM_BROWSER2_PROTOCOL_GUID{0xb9d4c360, 0xbcfb, 0x4f9b, {0x92, 0x98,
    ↪ 0x53, 0xc1, 0x36, 0x98, 0x22, 0x58}}
2 #define EFI_HII_CONFIG_ROUTING_PROTOCOL_GUID{0x587e72d7, 0xcc50, 0x4f79, {0x82,
    ↪ 0x09, 0xca, 0x29, 0x1f, 0xc1, 0xa1, 0x0f}}
```

and conduct a search. Finally, we located qword A28 as follows:

```
1 if(qword_140(&qword_A28[2], 0i64, &qword_DCF8) < 0 || qword_140(&qword_9F8[2], 0
    ↪ i64, &qword_DD00) < 0)
2     return 0x8000000000000000Eui64;
```

and found that this conditional statement called this variable. Then, we arranged the GUID of EFI HII CONFIG ROUTING PROTOCOL as d7 72 7e 58 50 cc 79 4f 82 09 ca 29 1f c1 a1 and searched for it, which was qword 9F8. Therefore, this conditional statement called both protocols. Returning to the original function, we found the relevant strings related to Secure Boot as shown in Figure 19.

```
.text:0000000000001B48 aSecureboot: ; DATA XREF: sub_73A0+8A4o
.text:0000000000001B48 ; sub_73A0+A44o
.text:0000000000001B48 text "UTF-16LE", 'SecureBoot',0
.text:0000000000001B5E align 20h
.text:0000000000001B60 aSetupmode: ; DATA XREF: sub_73A0+CE4o
.text:0000000000001B60 ; sub_73A0+E84o ...
.text:0000000000001B60 text "UTF-16LE", 'SetupMode',0
.text:0000000000001B74 align 8
```

Figure 19. Secure Boot Strings.

5. Discussion

One attack method on UEFI targets its settings. Once we acquire certain sensitive information, like the location of the password in NVRAM, it can be erased via the UEFI shell. Consequently, the UEFI firmware is left unprotected and can be altered at will. Based

on HII, it is easy to identify some critical codes, which should be given attention during the UEFI firmware design process.

In the above text, we only analyzed AMI firmware. Even though there are different manufacturers producing UEFI, such as Bysoft and Insyde, since the UEFI UI is developed based on the same HII framework, the UEFI firmware produced by different manufacturers is largely similar at the level of the UEFI UI. Therefore, the analysis of AMI firmware is highly representative. We still provide examples of password failure lockout times for UEFI produced by Bysoft and Insyde manufacturers as shown in Figures 20 and 21. For UEFI firmware produced by Bysoft and Insyde manufacturers, we can see that the password failure lockout times are directly displayed in plaintext without any protective measures. On the contrary, we need to spend a lot of effort to find the password failure lockout times in AMI's firmware.

3EB70	74 00 61 00 74 00 75 00 73 00 00 00 14 45 00 6E	t.a.t.u.s....E.n
3EB80	00 74 00 65 00 72 00 20 00 69 00 6E 00 63 00 6F	.t.e.r. .i.n.c.o
3EB90	00 72 00 72 00 65 00 63 00 74 00 20 00 70 00 61	.r.r.e.c.t. .p.a
3EBA0	00 73 00 73 00 77 00 6F 00 72 00 64 00 20 00 33	.s.s.w.o.r.d. .
3EBB0	00 20 00 74 00 69 00 6D 00 65 00 73 00 00 00 14	. .t.i.m.e.s....
3EBC0	50 00 6C 00 65 00 61 00 73 00 65 00 20 00 72 00	P.l.e.a.s.e. .r.
3EBD0	65 00 73 00 74 00 61 00 72 00 74 00 20 00 73 00	e.s.t.a.r.t. .s.
3EBE0	79 00 73 00 74 00 65 00 6D 00 00 00 14 50 00 72	y.s.t.e.m....P.r
3EBF0	00 65 00 73 00 73 00 20 00 45 00 73 00 63 00 20	.e.s.s. .E.s.c.
3EC00	00 66 00 6F 00 72 00 20 00 62 00 6F 00 6F 00 74	.f.o.r. .b.o.o.t
3EC10	00 20 00 6F 00 70 00 74 00 69 00 6F 00 6E 00 73	. .o.p.t.i.o.n.s

Figure 20. Password Attempt Times of Bysoft.

1750	00 00 00 00 70 00 61 00 73 00 73 00 77 00 6F 00	...p.a.s.s.w.o.
1760	72 00 64 00 73 00 20 00 74 00 68 00 72 00 65 00	r.d.s. .t.h.r.e.
1770	65 00 20 00 74 00 69 00 6D 00 65 00 73 00 20 00	.t.i.m.e.s. .
1780	75 00 73 00 69 00 6E 00 67 00 20 00 69 00 6E 00	u.s.i.n.g. .i.n.
1790	63 00 6F 00 72 00 72 00 65 00 63 00 74 00 20 00	c.o.r.r.e.c.t. .
17A0	63 00 72 00 65 00 64 00 65 00 6E 00 74 00 69 00	c.r.e.d.e.n.t.i.
17B0	61 00 6C 00 73 00 2E 00 00 00 00 00 00 00 00	a.l.s.....
17C0	00 00 00 00 59 00 6F 00 75 00 20 00 6D 00 75 00	...Y.o.u. .m.u.
17D0	73 00 74 00 20 00 72 00 65 00 73 00 74 00 61 00	s.t. .r.e.s.t.a.
17E0	72 00 74 00 20 00 74 00 68 00 65 00 20 00 63 00	r.t. .t.h.e. .c.
17F0	6F 00 6D 00 70 00 75 00 74 00 65 00 72 00 20 00	o.m.p.u.t.e.r. .
1800	74 00 6F 00 20 00 74 00 72 00 79 00 20 00 61 00	t.o. .t.r.y. .a.
1810	67 00 61 00 69 00 6E 00 2E 00 00 00 00 00 00	g.a.i.n.....

Figure 21. Password Attempt Times of Insyde.

All UEFI firmware human–machine interaction systems are developed based on HII. Therefore, if an attacker finds a vulnerability related to HII, it could affect all UEFI firmware. Hence, reverse engineering UEFI firmware can help security researchers discover and fix security vulnerabilities, thereby enhancing system protection capabilities.

In the real world, this knowledge will heighten the security awareness of UEFI interaction system developers. Often, developers have a weak sense of security concerning their developments. Understanding the working mechanism of UEFI firmware helps technical support personnel to diagnose and resolve firmware-related issues more effectively.

The potential benefit is the ability to improve the security of UEFI during its use, identifying and fixing potential security vulnerabilities, which can reduce the maintenance costs for UEFI manufacturers. The risk is that it might lead to the exposure of security vulnerabilities, increasing the risk of malicious exploitation.

In terms of Legal Considerations, UEFI firmware may be protected under copyright law and reverse engineering could infringe upon the intellectual property rights of the creators or owners of the firmware. It is crucial to review the licensing agreement of the firmware to understand what activities are permitted under its terms. Regarding Ethical Considerations, even though reverse engineering can benefit security enhancement and education, it is also essential to respect the labor and rights of the original creators.

Researchers should clarify their intentions when conducting reverse engineering to ensure the legitimacy of their actions and take responsibility for the potential consequences.

Therefore, before starting reverse engineering, confirm that the activity does not violate relevant laws, regulations, and licensing agreements. Evaluate the potential security risks that reverse engineering could bring, such as inadvertently creating new security vulnerabilities. Ensure legal compliance and use the acquired knowledge responsibly, but not for illegal or unethical purposes.

This article demonstrates that it is possible to extract critical information from UEFI via the HII framework. In the real world, there have been incidents of UEFI critical information leakage. Following a successful cyber-attack by the Money Message extortion group against MSI (Micro-Star International), some of MSI's internal documents were leaked, including sensitive data such as the BootGuard private key. The BootGuard security mechanism is an important part of Intel's hardware trust architecture, and the leakage of its private key indicates that for certain models of devices, this major security mechanism can be bypassed. Moreover, the leaked materials also included UEFI firmware image signing keys. BootGuard is a crucial integrity protection mechanism in Intel's security system based on the hardware root of trust, which, together with the CPU microcode and CSME, forms the core security mechanisms of the Intel ecosystem. If BootGuard is successfully attacked, or if attackers obtain the BootGuard private key from OEM/ODM manufacturers, they can exploit UEFI implementation flaws, underlying configuration errors, and other vulnerabilities to bypass or counter nearly all known security mechanisms. These mechanisms include system defenses such as BWE/BLE, PRx hardware security mechanisms, SecureBoot, HCVI, PatchGuard, kASLR, KDEP, SMEP, SMAP, and other kernel security protection technologies, as well as various mainstream antivirus software and EDR/XDR systems, achieving persistent control over the device.

UEFI is essentially a loader used to boot operating systems. An open ecosystem and more comprehensive functionality have allowed UEFI to replace the traditional BIOS. After nearly 20 years of development, UEFI has a vast amount of code, making it very difficult to understand. According to statistics, UEFI's share of the BIOS market started to decline annually after reaching 94%. CoreBoot has emerged from the Chromebook laptop market and has occupied a portion of the IoT market, always eyeing the server BIOS. Meanwhile, LinuxBoot has taken a different path, aggressively poaching from the server BIOS, and hopes to secure a foothold in the server market before descending to attack the PC, laptop, and IoT markets. Therefore, in addition to UEFI, there are many free, open-source loaders available such as coreboot, libreboot, linuxboot, and u-boot. When producing computers or servers, companies can choose the appropriate loader based on their needs to ensure that the product is sufficiently secure.

Subsequent research will focus on how to use these open-source loaders to ensure system security and that passwords saved within these open-source loaders are stored inside a secure hardware enclave, if available, or by using state-of-the-art cryptography to prevent cracking.

6. Related Work

6.1. Security Research on UEFI

Researchers Zimmer and Vincent first proposed that before executing the UEFI firmware executable file, its digital signature should be verified to ensure the integrity and legitimacy of the UEFI firmware executable file [32]. This method is based on digital authentication technology and provides a set of signing, authentication, and key management service interfaces to assist users, firmware systems, and operating systems in executing trustworthy operations. However, implementing this method relies on a large amount of authentication information and key information, which burdens the UEFI firmware memory; verifying the integrity of the file before execution affects the boot efficiency of the UEFI firmware [33]. Building upon Zimmer and Vincent's work, the literature [34] proposed a method based on trusted computing. Using the TPM chip as a trusted root node,

it establishes a trusted chain for computer system startup to ensure the legitimacy of the executing nodes, thereby securing the UEFI firmware boot process. This method establishes a trusted node on the hardware, starting from the platform power-on to the application execution, authenticating each level and trusting each level, building a complete trust chain, prohibiting the execution of unauthenticated programs, and identifying “self” and “non-self” components in a timely manner, thus constructing a secure computer operating environment. The challenge of this method is how to distinguish between the “self” and “non-self” components. Flexible distinction rules facilitate UEFI firmware applications, but they reduce the security of UEFI firmware. Strict rules ensure UEFI firmware security, but restrict its application, reducing firmware extensibility.

Attacks targeting UEFI are also frequently researched now. SMM (System Management Mode) is a CPU execution mode with the highest privileges, and attackers can modify the UEFI configuration and hardware chip using SMM vulnerabilities [32]. Moreover, attackers may exploit the S3 Resume Boot Script [35], Secure Boot [36,37], UEFI updates, and tamper with UEFI variables, among other tactics. In the face of increasingly severe firmware attacks, analyzing the vulnerability security of UEFI is of significant importance [38]. The paper first introduces the commonly used trusted authentication mechanisms of UEFI. Subsequently, targeting the vulnerabilities in the trust verification process of UEFI during the startup phase, and by integrating the state transition diagrams, the PageRank algorithm, and the Bayesian network theory, an analysis model for the UEFI trust verification startup vulnerability is constructed. The paper also conducts validation and analysis based on specific instances. Through the verification and analysis of the acquired data, vulnerable attack paths and critical vulnerability nodes are identified. Finally, based on the analysis results, security reinforcement measures for UEFI are proposed.

In addition to UEFI firmware for PCs and servers, the firmware of other platforms is also worthy of study. As a critical part in mobile cloud computing, the vulnerability of Android devices can directly affect the security of the mobile cloud. The unsecured Android can be potentially exploited by malwares to obtain the root privilege. Root privilege misuse is the critical issue for Android security, which breaks the integrity of Android security and rises the risk of permission escalation from malwares. The existing solutions still fail to balance the trade-off between the users desires on using root privilege and the Android security, which lays risks in leading to the root privilege misuse. To address this issue, Ref. [39] proposes a root privilege management scheme named Root Privilege Manager (RPM), which adopts root privilege access control to guarantee the exclusive root access opportunity of authenticated apps. RPM verifies the authorization and integrity of root-requesting apps based on the authorization files extracted during app installation and then controls the granting of root privilege based on these authenticated results. In this manner, end users are relieved of the burden of making appropriate decisions when confronted with root access management.

6.2. Reverse Analysis of UEFI

UEFI Image Reverse Engineering is extremely challenging. Some manufacturers use obfuscation or encryption techniques within the UEFI image to protect their code from being easily understood and modified. Additionally, UEFI images usually contain a large amount of complexly structured code. They might encompass numerous different modules, each implementing specific functions. Understanding how each module interacts with others, as well as the overall execution flow, demands an in-depth understanding and considerable effort. In addition, UEFI firmware is designed for specific hardware platforms and likely includes dependencies on specific hardware characteristics. Without knowledge relevant to the specific hardware platform, it may be difficult to comprehend the purpose and behavior of some code. UEFI applications are stored in the PE/COFF (Portable Executable/Common Object File Format), a complex binary format. Understanding the details of this format is crucial for analyzing UEFI firmware. While some tools can assist in the reverse engineering of UEFI firmware, they are often imperfect and require a deep

understanding of the UEFI architecture for effective usage. Furthermore, regarding some detailed implementation details and features of UEFI, relevant documentation may not be sufficient. In the article [40], the author proposed tracing the use of the LocateProtocol function. By analyzing the function's parameters and call relationships, it simplifies reverse analysis of UEFI firmware by understanding how the program interacts with various UEFI services. However, we still need an in-depth understanding and experience of UEFI, the PE/COFF format, and fundamental reverse engineering skills. Starting from 2014, the CHIPSEC framework [41] provides a set of modules, including hardware protection and the correct configuration test, which tests for vulnerabilities in firmware and platform components, fuzzing tools for various platform devices, and many more.

7. Conclusions

In this paper, we study the implementation details of the UEFI UI by reverse analyzing UEFI binary images, including UEFI password policy, sensitive information storage locations, and hardware port settings. We are the first to systematically apply reverse analysis techniques to study UEFI via the HII framework.

In terms of UEFI password policy, we found that the implementation methods vary among different manufacturers. By proposing a password policy analysis method based on the standard interface of AMI firmware, we provide a new approach for analyzing UEFI password policies. Moreover, we have identified storage locations for sensitive information related to the UEFI UI, offering a deeper understanding of the security of UEFI firmware. Regarding hardware port configurations, we analyzed how UEFI firmware utilizes the services provided by the HII framework to configure hardware ports.

Our research fills a gap in the study of the HII. Despite the achievements of our research, there are still many areas that need further exploration. In future research, we will continue to focus on the security and performance optimization of UEFI firmware to help manufacturers and users achieve safer and more efficient computing platforms. At the same time, we will also study more reverse engineering techniques and tools to gain a deeper understanding of the implementation mechanisms of UEFI firmware. In summary, this research provides a beneficial starting point for the field of UEFI firmware security research.

Author Contributions: Conceptualization, S.C., Y.-A.T., Z.Z. and Y.L.; Methodology, S.C., Y.L. and Q.Z.; Software, Y.-A.T.; Validation, S.C.; Formal analysis, S.C. and Q.Z.; Investigation, S.C., K.Q., Z.Z. and Y.L.; Resources, S.C. and K.Q.; Data curation, S.C.; Writing—original draft, S.C.; Writing—review & editing, S.C., Y.-A.T., K.Q., Y.L. and Q.Z.; Supervision, Y.-A.T., K.Q., Z.Z., Y.L. and Q.Z.; Project administration, Y.-A.T., Kefan Qiu, Y.L. and Q.Z.; Funding acquisition, Y.-A.T., Y.L. and Q.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the National Natural Science Foundation of China (no. U1936218, no. 62072037) and the China National Key Research and Development Program (no. 2022YFB2701501).

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. UEFI Specification. Available online: <https://uefi.org/specifications> (accessed on 8 October 2023).
2. Zimmer, V.; Rothman, M.; Marisetty, S. *Beyond BIOS: Developing with the Unified Extensible Firmware Interface*; Walter de Gruyter GmbH & Co KG: Berlin, Germany, 2017.
3. Jeong, D.; Lee, S. Forensic signature for tracking storage devices: Analysis of UEFI firmware image, disk signature and windows artifacts. *Digit. Investig.* **2019**, *29*, 21–27. [CrossRef]
4. Butterworth, J.; Kallenberg, C.; Kovah, X.; Herzog, A. Bios chronomancy: Fixing the core root of trust for measurement. In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, Berlin, Germany, 4–8 November 2013; pp. 25–36.
5. Sutherland, J.A. On Improving Cybersecurity Through Memory Isolation Using Systems Management Mode. Ph.D. Thesis, Abertay University, Dundee, Scotland, 2019.

6. Banik, S.; Zimmer, V. Understanding the BIOS and Minimalistic Design. In *System Firmware: An Essential Guide to Open Source and Embedded Solutions*; Apress: Berkeley, CA, USA, 2022; pp. 145–211.
7. Skalsky, N.; Kirch, T.; Rickey, A.; Rothman, M.A. UEFI and the OEM and IHV Community. *Intel Technol. J.* **2011**, *15*, 40–67.
8. Leara, W.D. Language Applications for UEFI BIOS. Ph.D. Thesis, The University of Texas at Austin, Austin, TX, USA, 2014.
9. Wilkins, R.; Richardson, B. UEFI secure boot in modern computer security solutions. In Proceedings of the UEFI Forum, Nancy, France, 26 September 2013; pp. 1–10.
10. UEFITool. Available online: <https://github.com/LongSoft/UEFITool> (accessed on 20 July 2023).
11. IDA Pro. Available online: <https://www.hex-rays.com/products/ida/> (accessed on 20 July 2023).
12. Universal-IFR-Extractor. Available online: <https://github.com/donovan6000/Universal-IFR-Extractor> (accessed on 20 July 2023).
13. Chess, B.; McGraw, G. Static analysis for security. *IEEE Secur. Priv.* **2004**, *2*, 76–79. [CrossRef]
14. Humphrey, W.S. *The Personal Software Process (sm)(PSP (sm))*; Carnegie Mellon University: Pittsburgh, PA, USA, 2000.
15. Vu, K.P.L.; Proctor, R.W.; Bhargav-Spantzel, A.; Tai, B.L.B.; Cook, J.; Schultz, E.E. Improving password security and memorability to protect personal and organizational information. *Int. J. Hum.-Comput. Stud.* **2007**, *65*, 744–757. [CrossRef]
16. Yan, J.J. A note on proactive password checking. In Proceedings of the 2001 Workshop on New Security Paradigms, Cloudcroft, NM, USA, 11–13 September 2001; pp. 127–135.
17. Summers, W.C.; Bosworth, E. Password policy: The good, the bad, and the ugly. In Proceedings of the Winter International Symposium on Information and Communication Technologies, Cancun, Mexico, 5–8 January 2004; pp. 1–6.
18. Polstra III, R.M. A case study on how to manage the theft of information. In Proceedings of the 2nd Annual Conference on Information Security Curriculum Development, New York, NY, USA, 23–24 September 2005; pp. 135–138.
19. Kuo, C.; Romanosky, S.; Cranor, L.F. Human selection of mnemonic phrase-based passwords. In Proceedings of the Second Symposium on Usable Privacy and Security, Pittsburgh, PA, USA, 12–14 July 2006; pp. 67–78.
20. Schwalb, D.; Berning, T.; Faust, M.; Dreseler, M.; Plattner, H. nvm malloc: Memory Allocation for NVRAM. *Adms@ VLDB* **2015**, *15*, 61–72.
21. Kiltz, S.; Altschaffel, R.; Dittmann, J. Hidden in Plain Sight-Persistent Alternative Mass Storage Data Streams as a Means for Data Hiding With the Help of UEFI NVRAM and Implications for IT Forensics. In Proceedings of the 2022 ACM Workshop on Information Hiding and Multimedia Security, Santa Barbara, CA, USA, 27–28 June 2022; pp. 107–112.
22. Lin, K.J.; Wang, C.Y. Using TPM to improve boot security at BIOS layer. In Proceedings of the 2012 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, USA, 13–16 January 2012; pp. 376–377.
23. Zhu, R.; Tan, Y.-A.; Zhang, Q.; Li, Y.; Zheng, J. Determining image base of firmware for ARM devices by matching literal pools. *Digit. Investig.* **2016**, *16*, 19–28. [CrossRef]
24. Zhu, R.; Zhang, B.; Mao, J.; Zhang, Q.; Tan, Y.-A. A methodology for determining the image base of ARM-based industrial control system firmware. *Int. J. Crit. Infrastruct. Prot.* **2017**, *16*, 26–35. [CrossRef]
25. Redini, N.; Machiry, A.; Wang, R.; Spensky, C.; Continella, A.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 1544–1561.
26. Davidson, D.; Moench, B.; Ristenpart, T.; Jha, S. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13), Washington, DC, USA, 14–16 August 2013; pp. 463–478.
27. Feng, B.; Mera, A.; Lu, L. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Berkeley, CA, USA, 12–14 August 2020; pp. 1237–1254.
28. Elsabagh, M.; Johnson, R.; Stavrou, A.; Zuo, C.; Zhao, Q.; Lin, Z. FIRMSCOPE: Automatic uncovering of Privilege-Escalation vulnerabilities in Pre-Installed apps in android firmware. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Berkeley, CA, USA, 12–14 August 2020; pp. 2379–2396.
29. Yin, J.; Li, M.; Wu, W.; Sun, D.; Zhou, J.; Huo, W.; Xue, J. Finding SMM Privilege-Escalation Vulnerabilities in UEFI Firmware with Protocol-Centric Static Analysis. In Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–26 May 2022; pp. 1623–1637.
30. Haładyn, K.M. An Extension for Ghidra That Adds Support for Reverse Engineering of UEFI Modules. Master’s Thesis, Instytut Telekomunikacji, Wrocław, Poland, 2021.
31. Rohleder, R. Hands-on ghidra—A tutorial about the software reverse engineering framework. In Proceedings of the 3rd ACM Workshop on Software Protection, London, UK, 15 November 2019; pp. 77–78.
32. Embleton, S.; Sparks, S.; Zou, C. SMM rootkits: A new breed of OS-independent malware. In Proceedings of the 4th International Conference on Security and Privacy in Communication Networks, Istanbul, Turkey, 22–25 September 2008; pp. 1–12.
33. Zimmer, V. Platform Trust Beyond BIOS Using the Unified Extensible Firmware Interface. In Proceedings of the Security and Management, Las Vegas, NV, USA, 25–28 June 2007; pp. 400–405.
34. Zhou, Z.L.; Xu, R.S. BIOS security analysis and a kind of trusted BIOS. In *Information and Communications Security, Proceedings of the 9th International Conference, ICICS 2007, Zhengzhou, China, 12–15 December 2007*; Springer: Berlin/Heidelberg, Germany, 2007; pp. 427–437.

35. Bazhaniuk, O.; Bulygin, Y.; Furtak, A.; Gorobets, M.; Loucaides, J.; Matrosov, A.; Shkatov, M. Attacking and Defending BIOS in 2015. In Proceedings of the ReCon Conference, Montreal, QC, Canada, 19–21 June 2015. Available online: <http://www.intelsecurity.com/advanced-threat-research/content/AttackingAndDefendingBIOS-RECon2015.pdf> (accessed on 8 October 2023).
36. Bulygin, Y.; Furtak, A.; Bazhaniuk, O. *A Tale of One Software Bypass of Windows 8 Secure Boot*; Black Hat: Las Vegas, NV, USA, 2013.
37. Jacob, N.; Heyszl, J.; Zankl, A.; Rolfes, C.; Sigl, G. How to break secure boot on fpga socs through malicious hardware. In *Cryptographic Hardware and Embedded Systems—CHES 2017, Proceedings of the 19th International Conference, Taipei, Taiwan, 25–28 September 2017*; Springer: Cham, Switzerland, 2017; pp. 425–442.
38. Gu, Y.; Zhang, P.; Chen, Z.; Cao, F. UEFI Trusted Computing Vulnerability Analysis Based on State Transition Graph. In Proceedings of the 2020 IEEE 6th International Conference on Computer and Communications (ICCC), Chengdu, China, 11–14 December 2020; pp. 1043–1052.
39. Tan, Y.-A.; Xue, Y.; Liang, C.; Zheng, J.; Zhang, Q.; Zheng, J.; Li, Y. A root privilege management scheme with revocable authorization for Android devices. *J. Netw. Comput. Appl.* **2018**, *107*, 69–82. [[CrossRef](#)]
40. Lebedev, P.; Kogos, K.; Vasilenko, E. On Way to Simplify the Reverse Engineering of UEFI Firmwares. In *Advances in Cyber Security, Proceedings of the First International Conference, ACeS 2019, Penang, Malaysia, 30 July–1 August 2019*; Springer: Singapore, 2020; pp. 220–231.
41. Chipsec. Available online: <https://github.com/chipsec/chipsec> (accessed on 8 October 2023).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.