Design and development of an automated regression test suite for UEFI

Master Thesis

Leading to a master’s degree in
Automotive Software Engineering
Professorship of Computer Engineering
Faculty of Computer Science

Submitted by: Huzaifa Saadat
Supervisors: Prof. Dr. Wolfram Hardt
          Mr. Bertram Metz
          Mr. Robert Fendt

Augsburg, October 2014
Declaration of Authorship

I, Huzaifa Saadat, assure that the thesis “Design and development of an automated regression test suite for UEFI” is my own work under the guidance of my supervisors. The data collected during the literature review and referred in this document is given due acknowledgement. All the references and helping materials are enlisted in the Bibliography with all sincerity.

Signature: _____________________

Date: _________________________
Abstract
Unified Extensible Firmware Interface (UEFI) is an industry standard for implementing the basic firmware in the computers. This standard replaces BIOS. A huge amount of C code has been written for the implementation of UEFI. Yet there has been a very little focus on testing UEFI code. The thesis shows how the industry can perform a meaningful testing of UEFI. Spanning the test coverage with the help of test tools over all UEFI phases is a key objective. Moreover, techniques such as Test Driven Development and source code analysis are explained in terms of UEFI to make sure the bugs are minimized in the first place. The results show that the usage of test and analysis tools point to a large number of issues. Some of these issues can be fixed at a very early stage in the Software Development Life Cycle. For this reason the developers and testers should be convinced that they need to focus on testing UEFI from a software perspective.
Acknowledgements

In the name of Allah, the Most Gracious, the Most Merciful.

I start off by thanking Mr. Robert Fendt, my Technical Supervisor at General Electric Intelligent Platforms (GE IP). After the first few discussions I knew I was in the right hands to accomplish something as complex as a Master Thesis. His valuable suggestions, regular meetings and dedicated time led me to the successful completion of this thesis. His phrase at the end of every meeting “Wow! Good work” kept me going through the ups and downs of my research. Thanks to his aggressive approach, a thesis that started as a testing centered ended up with over all code quality improvement.

Secondly, I would like to thank Mr. Bertram Metz, my Managerial Supervisor at GE IP. It was his initiative that led to the idea of this topic. During my thesis he ensured I had everything that was needed to complete it successfully.

Moving to the academic side, my Professor, Dr. Wolfram Hardt contributed to the development of my skills even before my thesis had started. These skills certainly helped me better understand the engineering problems and taught me the art of technical documentations. As my supervisor he had given me precise and valuable suggestions during the thesis, especially at the concept presentation. So, thank you Professor Wolfram Hardt.

Dr. Ariane Heller had been coordinating with me throughout the thesis. She had handled the official procedures so well that it gave me a feeling as if looking after my thesis was her only task (this obviously wasn’t her only task!). Her hints on my documentations and presentations were beyond helpful. Thanks for your time and effort.

There were two aspects of support during the course of my thesis; practical and moral. Having mentioned the practical support, and beginning with the moral support, I would like to thank my parents and siblings. Though they were miles away from me, their prayers reflected in the success I had along the thesis. At the start of the thesis, I made a commitment with myself of working on the thesis only during the office hours. It was their moral support that allowed me to keep this commitment, yet meet all the deadlines.
# Table of Contents

DECLARATION OF AUTHORSHIP .................................................................................. I

ABSTRACT .................................................................................................................. II

ACKNOWLEDGEMENTS .............................................................................................. III

LIST OF FIGURES AND TABLES .................................................................................. VI

ABBREVIATIONS .......................................................................................................... VII

1. INTRODUCTION ...................................................................................................... 1
   1.1 Motivation ........................................................................................................... 1
   1.2 Objective ........................................................................................................... 2
   1.3 Overview ........................................................................................................... 2

2. STATE OF THE ART ................................................................................................. 4
   2.1 Need for UEFI .................................................................................................... 4
   2.2 Phases in UEFI .................................................................................................. 7
   2.3 UEFI Specifications and Vendors (IBVs) ............................................................ 11
     2.3.1 UEFI/EDK Specifications ............................................................................ 12
     2.3.2 AMI ............................................................................................................. 16
     2.3.3 INSIDE ....................................................................................................... 17
     2.3.4 Phoenix ...................................................................................................... 17
   2.4 Lack of UEFI Testing .......................................................................................... 18
   2.5 Issues with testing UEFI ..................................................................................... 20
   2.6 Summary ............................................................................................................ 22

3. AVAILABLE UEFI TEST TOOLS ............................................................................. 24
   3.1 Firmware Test Suite .......................................................................................... 25
   3.2 Platform Initialization Self-Certification Test .................................................... 28
   3.3 CHIPSEC .......................................................................................................... 32
   3.4 Summary ............................................................................................................ 34

4. TEST DRIVEN DEVELOPMENT FOR UEFI .......................................................... 35
   4.1 Importance of Test Driven Development ......................................................... 36
   4.2 Unit Tests .......................................................................................................... 38
   4.3 Test Driven Development in Aptio V ................................................................. 40
   4.4 Summary ............................................................................................................ 44

5. CODE ANALYSIS FOR UEFI .................................................................................. 46
   5.1 Static Analysis .................................................................................................... 46
     5.1.1 PVS-STUDIO ............................................................................................. 47
     5.1.2 Cppcheck ................................................................................................... 50
     5.1.3 CppDepend ............................................................................................... 52
     5.1.4 Tools Comparison ....................................................................................... 57
   5.2 Dynamic Analysis .............................................................................................. 60
   5.3 Summary ............................................................................................................ 63

6. RESULTS ................................................................................................................... 64
   6.1 Results Criteria .................................................................................................. 64
     6.1.1 Available UEFI Test Tools Results .............................................................. 65
     6.1.2 Test Driven Development for UEFI Results .............................................. 66
     6.1.3 Code Analysis for UEFI Results ............................................................... 67
   6.2 Further Options ................................................................................................ 68

7. CONCLUSION .......................................................................................................... 70
List of Figures and Tables

**TABLE 1.1: ABBREVIATIONS** ............................................................................................................................. VIII
**FIGURE 2.1: UEFI boot phases [8]......................................................................................................................... 8
**FIGURE 2.2: UEFI runtime services [13]................................................................................................................ 11
**FIGURE 2.3: EDK II specific build process [19]..................................................................................................... 15
**FIGURE 2.4: VEB showing AMI and Intel implementations .................................................................................. 16
**FIGURE 3.1: COMPLETE COVERAGE OF UEFI PHASES USING THE TEST TOOLS........................................... 24
**FIGURE 3.2: FwTS Live [36].................................................................................................................................. 27
**FIGURE 3.3: PVS C SCT native mode menu-driven interface [41]........................................................................ 32
**FIGURE 3.4: Chipsec components and structure [42]............................................................................................ 33
**FIGURE 4.1: Debug Later Programming [44].......................................................................................................... 36
**FIGURE 4.2: Test Driven Development cycle [44]................................................................................................... 37
**FIGURE 4.3: Package structure in APTIO V.......................................................................................................... 41
**FIGURE 4.4: TDD Package structure in APTIO V.................................................................................................. 42
**FIGURE 5.1: PVS-Studio Analysis under progress................................................................................................. 47
**FIGURE 5.2: CPPDepend query rules and groups ................................................................................................ 53
**FIGURE 5.3: Dependency Graph.......................................................................................................................... 56
**FIGURE 5.4: Dependency Matrix ........................................................................................................................ 56
**FIGURE 5.5: Metrics............................................................................................................................................... 57
**FIGURE 5.6: Application with DR. Memory ........................................................................................................... 61
**FIGURE 6.7: Avatar architecture [59]................................................................................................................ 62
**TABLE 6.1: MANUAL TESTING VS AUTOMATED TESTING FOR UEFI .................................................................... 65
**TABLE 6.2: Static analysis tools results ................................................................................................................ 67
## Abbreviations

<table>
<thead>
<tr>
<th>Full form</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unified Extensible Firmware Interface</td>
<td>UEFI</td>
</tr>
<tr>
<td>Basic Input Output System</td>
<td>BIOS</td>
</tr>
<tr>
<td>Operating System</td>
<td>OS</td>
</tr>
<tr>
<td>EFI Development Kit</td>
<td>EDK (EDKII)</td>
</tr>
<tr>
<td>Trusted Platform Module</td>
<td>TPM</td>
</tr>
<tr>
<td>Motor Industry Software Reliability Association</td>
<td>MISRA</td>
</tr>
<tr>
<td>Power On Self-Test</td>
<td>POST</td>
</tr>
<tr>
<td>Independent BIOS Vendors</td>
<td>IBVs</td>
</tr>
<tr>
<td>Personal Computer</td>
<td>PC</td>
</tr>
<tr>
<td>Globally Unique Identifiers</td>
<td>GUIDs</td>
</tr>
<tr>
<td>Original Equipment Manufacturer</td>
<td>OEM</td>
</tr>
<tr>
<td>Personal Digital Assistants</td>
<td>PDAs</td>
</tr>
<tr>
<td>Master Boot Record</td>
<td>MBR</td>
</tr>
<tr>
<td>GUID Partition Table</td>
<td>GPT</td>
</tr>
<tr>
<td>Platform Initialization</td>
<td>PI</td>
</tr>
<tr>
<td>Security</td>
<td>SEC</td>
</tr>
<tr>
<td>Pre EFI Initialization</td>
<td>PEI</td>
</tr>
<tr>
<td>Driver Execution Environment</td>
<td>DXE</td>
</tr>
<tr>
<td>Boot Device Selection</td>
<td>BDS</td>
</tr>
<tr>
<td>Transient System Load</td>
<td>TSL</td>
</tr>
<tr>
<td>Run Time</td>
<td>RT</td>
</tr>
<tr>
<td>After Life</td>
<td>AL</td>
</tr>
<tr>
<td>PEI Modules</td>
<td>PEIM</td>
</tr>
<tr>
<td>Hand-Off Blocks</td>
<td>HOBs</td>
</tr>
<tr>
<td>Architectural Protocols</td>
<td>APs</td>
</tr>
<tr>
<td>Non-Volatile Random Access Memory</td>
<td>NVRAM</td>
</tr>
<tr>
<td>Human Interface Infrastructure</td>
<td>HII</td>
</tr>
<tr>
<td>PEIM to PEIM Interfaces</td>
<td>PPIs</td>
</tr>
<tr>
<td>Independent Hardware Vendors</td>
<td>IHVs</td>
</tr>
<tr>
<td>System Management Mode</td>
<td>SMM</td>
</tr>
<tr>
<td>System Management Bus</td>
<td>SMBus</td>
</tr>
<tr>
<td>Module Information</td>
<td>INF (.inf)</td>
</tr>
<tr>
<td>Package Declaration</td>
<td>DEC (.dec)</td>
</tr>
<tr>
<td>Platform Description</td>
<td>DSC (.dsc)</td>
</tr>
<tr>
<td>Flash Description File</td>
<td>FDF (.fdf)</td>
</tr>
<tr>
<td>Flash Device</td>
<td>FD</td>
</tr>
<tr>
<td>Firmware Volume</td>
<td>FV</td>
</tr>
<tr>
<td>Platform Configuration Databases</td>
<td>PCDs</td>
</tr>
<tr>
<td>American Megatrends Incorporation</td>
<td>AMI</td>
</tr>
<tr>
<td>Visual eBIOS</td>
<td>VeB</td>
</tr>
<tr>
<td>Independent Development Environment</td>
<td>IDE</td>
</tr>
<tr>
<td>Hardware-2-Operating System</td>
<td>H2O</td>
</tr>
<tr>
<td>Light Emitting Diodes</td>
<td>LEDs</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Software Development Life Cycle</td>
<td>SDLC</td>
</tr>
<tr>
<td>Firmware Test Suite</td>
<td>fwts</td>
</tr>
<tr>
<td>Platform Initialization Self-Certification Test</td>
<td>PI SCT</td>
</tr>
<tr>
<td>Linux UEFI Validation</td>
<td>LUV</td>
</tr>
<tr>
<td>Compact Disk</td>
<td>CD</td>
</tr>
<tr>
<td>Universal Serial Bus</td>
<td>USB</td>
</tr>
<tr>
<td>Advanced Configuration and Power Interface</td>
<td>ACPI</td>
</tr>
<tr>
<td>Distributed Management Task Force</td>
<td>DMTF</td>
</tr>
<tr>
<td>Desktop Management Interface</td>
<td>DMI</td>
</tr>
<tr>
<td>System Management BIOS</td>
<td>SMBIOS</td>
</tr>
<tr>
<td>Compatibility Support Module</td>
<td>CSM</td>
</tr>
<tr>
<td>Complementary Metal-Oxide Semiconductor</td>
<td>CMOS</td>
</tr>
<tr>
<td>Extended BIOS Data Area</td>
<td>EBDA</td>
</tr>
<tr>
<td>Task Priority Level</td>
<td>TPL</td>
</tr>
<tr>
<td>Visual Studio</td>
<td>VS</td>
</tr>
<tr>
<td>Driver Development Kit</td>
<td>DDK</td>
</tr>
<tr>
<td>UEFI Management Side</td>
<td>EMS</td>
</tr>
<tr>
<td>Hardware Abstraction Layer</td>
<td>HAL</td>
</tr>
<tr>
<td>Model Specific Registers</td>
<td>MSR</td>
</tr>
<tr>
<td>Interrupt Descriptor Table</td>
<td>IDT</td>
</tr>
<tr>
<td>Global Descriptor Table</td>
<td>GDT</td>
</tr>
<tr>
<td>SMM Range Register</td>
<td>SMRR</td>
</tr>
<tr>
<td>Test Driven Development</td>
<td>TDD</td>
</tr>
<tr>
<td>Standard Template Library</td>
<td>STL</td>
</tr>
<tr>
<td>Software Testing Automation Framework</td>
<td>STAF</td>
</tr>
<tr>
<td>STAF eXecution engine</td>
<td>STAX</td>
</tr>
</tbody>
</table>

Table 1.1: Abbreviations
Chapter 1

1. Introduction
UEFI stands for Unified Extensible Firmware Interface. It is the firmware in computers that ensures initialization of hardware resources and setting up of the environment for Operating Systems (OS). It replaces the legendary Basic Input Output System (BIOS). The standard is relatively new and hasn’t gone through much testing.

This thesis focuses on improving the quality of UEFI. To achieve this ultimate goal, the thesis was divided into 3 phases. The first phase was to use the already available test tools and maximize the test coverage. UEFI is a huge project and hence the traditional black-box oriented manual testing is unfeasible. As opposed to manual testing, the test tools are automatic. The second phase is visualizing Test Driven Development (TDD) for UEFI which focuses on minimizing the bugs in the first place. The target audience for this approach is the UEFI/BIOS developers. The third phase is usage of code analysis techniques to improve the C code for UEFI implementation. There are two code analysis approaches; static code analysis and dynamic code analysis. Static code analysis performs the analysis without executing the code whereas dynamic code analysis performs the analysis while executing the code. A comparison and usefulness of 3 static analysis tools, with respect to UEFI, are provided. An idea on how to perform dynamic analysis for UEFI is also presented.

1.1 Motivation
Over a long period of time, BIOS has started showing its weaknesses and became unmanageable. BIOS was meant for a specific purpose and was forcefully scaled/modifed to be used beyond its scope. At last BIOS succumbed to the ever-growing modifications. The industry would not want UEFI to follow the same path.

The best way to ensure that UEFI remains on the right track is to grill it during the early stage of its life. Now is the time to point out any showstoppers UEFI has. To be able to do this, we must test it thoroughly and fix the issues as early as possible. Similarly UEFI should go through regression tests to gain confidence during the code changes and new feature implementations.
While criticizing UEFI, Matthew Garrett said “the only people to enable UEFI are enthusiasts.” However, he also said “after a few years of iterative improvements it (UEFI) stands a good chance of being more reliable and useful than BIOS.” [1] The real motivation behind this thesis is to make sure that UEFI reaches its maturity as soon as possible.

1.2 Objective
The first and foremost objective of this thesis is to make the industry realize the importance of testing UEFI. On the way to achieve this goal, misconceptions have to be removed. One of the biggest misconceptions is to view hardware diagnostic tools as UEFI confidence building tests. As the saying goes, the devil is in the detail. Hardware diagnostics will not point out the bulk of issues in the code. Thus, UEFI has to be tested from a software perspective. Another misconception is to be satisfied on the quality of UEFI once manual tests have been successfully executed. There is much more to UEFI than what manual black-box tests can show.

It is important to practically show that one test tool may not be sufficient to cover all phases in UEFI. Thus a combination of tools should be used such that they are neither much overlapping nor leave a huge gap. This is another important objective.

Only the testers should not be responsible for ensuring the quality of UEFI. The UEFI/BIOS developers can ensure minimization of bugs by implementing Approaches such as Test Driven Development and code analysis.

1.3 Overview
There is a lot of work being done on UEFI. Some of the existing work has been focused on securing UEFI. Security should not be confused with testing. Secure Boot is a security feature in UEFI. This feature, like other features, has to be extensively tested. Trusted Platform Module (TPM) is once again a security feature that has been spoken of in many researches. Again this is a feature and not be perceived as test ensuring UEFI code quality. This thesis focuses on testing UEFI code.
There is a book that talks about Test driven Development for Embedded C. My research included the understanding and implementation of Test Driven Development specifically for UEFI. This should be the only piece of literature talking about Test Driven Development for UEFI, as of now.

The shift from BIOS to UEFI brought another advantage i.e. the programming language shifted from assembly to C. With this change we should try to take the advantage of the well-established C language. One such advantage is the usage of code analysis tools. There are lots of static code analysis tools for C. Likewise many studies have shown a comparison of static analysis tools. TERA-Labs conducted a research that compared the static code analysis tools for MISRA-C rules. My research showed how useful the static analysis tools can be for UEFI, considering the fact that the implementation contains metadata files apart from the C source files and there are no standard C libraries in the UEFI project.

I have structured my thesis such that the first chapter gives a brief introduction, motivation, objectives and states the difference between my research and other researches. The second chapter provides a prerequisite knowledge to understand the work in later chapters. It explains the need and phases in UEFI. The chapter further gives the state of the art or the current industry scenario/problems. The third chapter talks about how 3 test tools can be used to cover testing in UEFI. The fourth chapter explains how Test Driven Development was implemented under the UEFI environment. The fifth chapter discusses code analysis tools and techniques. It gives an overview of how to visualize these tools and techniques for UEFI. The final chapter discusses the results of my thesis along with the concluding remarks. Ideas on further research possibilities are also mentioned in the sixth chapter.

The chapters/headings are divided in up to 3 levels e.g. 2.3.4. The names of tools, files (with extensions) and UEFI packages are written in italic. The names of functions are written in Courier New. First usage of an abbreviation is written along with its full form. The caption of a figure or a table briefly explains it. The quotations are given in “quotation marks”. These aesthetic modifications are meant for reader friendliness.
Chapter 2

2. State of the Art

This chapter describes the state of the art. In other words it should give the reader a good insight as to the basic understanding of UEFI and the current industry scenario. The sub sections will explain the need for replacing legendary BIOS with UEFI. A brief discussion on phases in UEFI will be presented. UEFI specification and some of the UEFI/BIOS vendors will be introduced. Finally the current UEFI testing scenario and its problems will be mentioned.

2.1 Need for UEFI

The prerequisite to understand the need for UEFI is to understand the problems with BIOS. The basic concept of BIOS was to test the system and provide input, output and boot devices initialization. BIOS had two major functions; running Power On Self-Test (POST) and provide an abstraction layer for the Operating System (OS) i.e. runtime. POST could be considered as a diagnostic test. The information provided by POST is either beep codes or POST codes. [2] POST codes are a two digit hex number, indicating the progress of POST. The beep code on the other hand is specific to the Independent BIOS Vendor (IBV). The number of times a beep is sounded during the boot process indicates a specific error. Normally, if there isn’t any problem the user may hear only one beep.

For about three decades BIOS seemed to work well as a firmware for the Personal Computers (PC). At last BIOS succumbed to the ever fast growing advancement in this field and started showing its limitations. Firstly, the BIOS had poor synchronization with the hardware. As a result BIOS had to be poll driven instead of being interrupt driven. Secondly, BIOS assumed the processor to be in real mode. This assumption became redundant due to the operating systems that did not run in real mode. The real mode was based on 8086 and 8088 processors which had 16 bit instructions. This was true for the DOS and early Windows era but not for the modern era which has 32 and 64 bit versions of OS. [3] Another drawback, worth noting, of BIOS was its lack of extensibility. The abstraction provided by BIOS failed to cope with the modernization
in technology. However, the attempt to cope with new technologies was made by implementing specific upgrades for a specific problem. This meant that the problem resolution for one platform may not be a solution for another platform. Yet the most significant drawback in context of present time was that the BIOS implementations were largely done in assembly language. There were several other issues such as lack of well-defined standardization of Option ROMs. Option ROMs has a similar concept to device drivers i.e. the access of input output devices by the basic software, which are unknown by default. However, the difference with respect to device drivers is that the Option ROMs is a firmware terminology rather than an OS terminology. Furthermore, companies in the PC business saw BIOS as a place to earn their competitive advantage. Thus, inevitably most critical implementations were done in BIOS. This led the BIOS to become over-crowded or made the architecture very poor. Finally, the work around was no longer a sane possibility to extend 16-bit code to 64-bit Itanium instructions. [4]

All these problems meant that a need for a new firmware standard arose. This new standard happened to be Extensible Firmware Interface (EFI) by Intel. The standard was handed over to Unified Extensible Firmware Interface Forum and thus became known as UEFI. The goal of UEFI was not only to address the issues of BIOS but also to preempt possible limitations in the near future and act upon it to ensure that UEFI too, at least, lives as long as BIOS did. This meant that while addressing the issues of BIOS, the well thought out advantages of BIOS should be carried along.

One such advantage of BIOS has been its neutrality towards OS, which has been carried along as part of UEFI implementations. The major win of the UEFI standard is its capability to be extensible. The importance of this goal can be understood from the fact that the word “Extensible” is part of the standard’s name (Unified “Extensible” Firmware Interface). This goal has been achieved by introducing the concept of Globally Unique Identifiers (GUIDs). Using GUIDs the companies/developers can implement their own interfaces and still ensure that a clash will not occur due to the previous implementation of Original Equipment Manufacturer (OEM). GUIDs are normally written as 32 hexadecimal digits; this means that there are $2^{128}$ unique possibilities. Several web applications are present to automatically generate a random GUID. This extensibility property was not available in BIOS. Another advantage of UEFI is its modularity property. Modules may be implemented by different
companies/developers and then neatly integrated into the whole project. This is made possible by EFI inter-module cooperation. An example of this advantage is that Intel implements a few functionalities IBVs may acquire that code and implement their own functionalities in a separate module without having much knowledge of Intel’s implementation. Likewise PC vendors may acquire that code from an IBV and implement their own functionalities in a separate module without having much knowledge of either Intel or IBV implementations. Moreover, modularity will indirectly also mean that the code may be re-used. Providing basic services commonly used by operating systems, such as priority levels, is another goal of UEFI. Furthermore, apart from being platform and OS independent UEFI takes a step forward and aims to be instruction set independent as well. Ideally, a UEFI implementation that works on a specific hardware with a specific instruction set should also work on the hardware with another instruction set.

As for the programmers a significant change from BIOS to UEFI is in the fact that the coding can now be done in high level language (C language) rather than in a low level language (Assembly language). Option ROMs in UEFI are considered as drivers with more or less the same privileges as other modules. This way the problem of lack of Option ROM standardization has been partially addressed in UEFI. The option ROMs had a limited storage and had to be upgraded separately if the hardware is upgraded, these issues theoretically stand resolved. [5] The issue of scalability has been addressed in UEFI as well. UEFI shall support all devices i.e. Personal Digital Assistants (PDAs), notebooks, laptops, PCs, servers etc. BIOS was meant for the PCs and had to be worked upon to scale it for other devices. Another interesting addition in UEFI is the Boot Manager. It has two major functions. Firstly, it acts as a centralization point for controlling boot order. Secondly, the optimizations of boot time are managed here. UEFI partially solves the problem of BIOS by accommodating interrupt driven concept. This was made possible by introducing Task Priority Levels. [4] The Master Boot Record (MBR) way of managing disks in BIOS has been replaced by UEFI. UEFI now uses GUID Partition Table (GPT). [6] MBR maintained the information about disk partitions and the location of a bootable device. With time the information/addresses/data became so large for MBR to keep track of them. Furthermore, MBR supported four partitions by default and required extended partition to cater more. Thus GPT, with a more flexible way of partitioning table has been
introduced. GPT allows many partitions to be formed. Each partition is identified by a unique GUID and thus removing a possibility of a collision of disk access. [7] The task of the OS boot loader is now taken over by UEFI. The OS is now wary of executing boot loading tasks. One such example is that the OS will not have to ask if the user likes to boot from a safe mode or not. [6] Lastly, the network can be accessed from UEFI preboot environment whereas; this was not possible in BIOS.

The awareness and popularity of UEFI has been growing. However, currently both (BIOS and UEFI) can be used in the system as a booting firmware. Here I should also mention that during the UEFI Plugfest 2014, Vincent Zimmer (Intel Corporation and co-author of Beyond BIOS) was asked how long will they continue the support for legacy BIOS? He answered “the rule of thumb at Intel was, after the last Microsoft OS that supports MBR (Master Boot Record) boot… 10 years from then.” This means that though the research is now focused on UEFI, it has not completely replaced BIOS in the market nor will it in the near future. UEFI seems to handle speed, functionality and adaptability better than BIOS. UEFI boots faster than BIOS when ran with optimized boot settings on the same hardware platform. [5]

2.2 Phases in UEFI
The Platform Initialization (PI) boot phase can be divided into 7 phases:

1. Security (SEC)
2. Pre EFI Initialization (PEI)
3. Driver Execution Environment (DXE)
4. Boot Device Selection (BDS)
5. Transient System Load (TSL)
6. Run Time (RT)
7. After Life (AL)
The first phase of UEFI is the Security (SEC) phase. This phase is triggered by a power-on or a reset. The SEC phase is responsible for ensuring the integrity of the firmware. It is platform and processor dependent. The implementation of this phase is most likely to be in assembly language and the amount of code here should not be much compared to the PEI and DXE phase. Amongst the responsibilities of SEC phase are:

1. All restart events are taken care of by the SEC phase. The CPU cache is cleaned and the initialization code is run from the ROM.
2. Creates temporary memory storage. This is achieved by setting a portion of the cache to a known state.
3. It is the root of trust in the system taking the initial control. SEC phase then becomes an authentication authority for the next phase i.e. PEI. The loaded or executed code is trusted or checked for integrity before hand over to PEI phase.
4. Finally, the control is handed over to the PEI foundation.
The second phase during PI is the Pre EFI Initialization (PEI) phase. As the name suggests, as well as what can be partially interpreted from the diagram above, PEI is responsible for initializing platform resources. This in turn helps DXE phase to execute in C environment. [9] PEI phase assists the system until the permanent RAM is available. [11] Amongst the tasks of PEI are:

1. Initialization of the processor making use of INIT command.

2. It is a small start-up code that transfers services running within ROM to CPU cache. [10]

3. It is responsible for locating, validating, dispatching and executing PEI modules (PEIM). [9] These modules are responsible for completing other tasks in PEI phase such as main memory initialization. These are chipset/platform specific, supporting their features.


5. Finally the platform information is handed over to DXE and the DXE core is launched. [9]

The third phase during PI is the Driver Execution Environment (DXE) phase. Here the devices are detected and initialized. The protocol and driver implementation is done in this phase. UEFI services and UEFI interface tables are made available after this phase. DXE helps the drivers to be in a modular fashion (similar to the way OS companies do). Historically this was very difficult because the boot flow was controlled by a single source file. The code written back then was hard to port from a platform to another. The DXE and UEFI drivers are dispatched during this phase. The DXE drivers are dispatched first and then the UEFI drivers. The difference between DXE drivers and UEFI drivers is that the DXE drivers may have specific dependency rules. [9] DXE phase executes from the time when memory is available to the system until the time firmware is ready to look for the boot devices. [11] The components within DXE are as follows: [9]

1. Drivers; which initialize chipset, processors and platform components.
2. Foundation; which is the DXE executable binary and is responsible for running services and dispatching drivers.

3. Architectural protocols (APs); which are called by the foundation and produced by the drivers.

4. EFI system tables; which points to the UEFI tables, handle data bases and console devices.

5. Dispatcher; which queues up the available drivers to be executed in the right order.

The fourth phase in PI is the Boot Device Selection (BDS) phase. The boot device is selected here and the OS is booted from it. The signature is verified provided the secure boot is turned on. If the boot loader is not signed UEFI will deny its execution. [10] The user may decide to go to the setup from here or run the shell, apart from booting a specific device. There are 6 steps in the BDS phase: [9]

1. Initialize the language for the end user and database to show the text during BDS.

2. Accumulate device list.

3. Connect the devices to UEFI drivers.

4. Detect console devices.

5. Perform memory test.

6. Process the boot devices.

The fifth phase is the Transient System Load (TSL). UEFI applications can be implemented to provide temporary services to the OS e.g. a utility to create partitions or perform system diagnostics. [12] Practically this could be visualized as the phase where the user did not boot the OS even though the boot loader was ready to do so. For instance, EFI Development Kit (EDK) allows an implementation of an UEFI application. This `.efi` application can run from the shell. While we are in the shell or while running a ‘hello world!’ UEFI application we would be in the TSL phase.

The sixth phase is the Runtime (RT). This is the phase where OS is now running and UEFI has completed its essential jobs. However, UEFI is now playing an OS supportive role. The runtime services are still active with the aim of abstracting hardware from the
OS. The boot services are now exited. The following table shows the UEFI runtime service function calls:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetTime()</td>
<td>Returns the current time, time context, and time keeping capabilities.</td>
</tr>
<tr>
<td>SetTime()</td>
<td>Sets the current time and time context.</td>
</tr>
<tr>
<td>GetWakeupTime()</td>
<td>Returns the current wakeup alarm settings.</td>
</tr>
<tr>
<td>SetWakeupTime()</td>
<td>Sets the current wakeup alarm settings.</td>
</tr>
<tr>
<td>GetVariable()</td>
<td>Returns the value of a named variable.</td>
</tr>
<tr>
<td>GetNextVariableName()</td>
<td>Enumerates variable names.</td>
</tr>
<tr>
<td>SetVariable()</td>
<td>Sets, and if needed creates, a variable.</td>
</tr>
<tr>
<td>SetVirtualAddressMap()</td>
<td>Switches all runtime functions from physical to virtual addressing.</td>
</tr>
<tr>
<td>ConvertPointer()</td>
<td>Used to convert a pointer from physical to virtual addressing.</td>
</tr>
<tr>
<td>GetNextHighMonotonicCount()</td>
<td>Subsumes the platform’s monotonic counter functionality.</td>
</tr>
<tr>
<td>ResetSystem()</td>
<td>Resets all processors and devices and reboots the system.</td>
</tr>
<tr>
<td>UpdateCapsule()</td>
<td>Passes capsules to the firmware with both virtual and physical mapping.</td>
</tr>
<tr>
<td>QueryCapsuleCapabilities()</td>
<td>Returns if the capsule can be supported via UpdateCapsule().</td>
</tr>
<tr>
<td>QueryVariableInfo()</td>
<td>Returns information about the EFI variable store.</td>
</tr>
</tbody>
</table>

Figure 2.2: UEFI runtime services [13]

The seventh and the last phase is the After Life (AL). There’s nothing specific that happens in this phase. It just completes the loop. The concept for this phase is that the UEFI should be prepared to take back control in case the OS fails or is exited. AL comes into play if the OS crashes or the system is sent to a sleep/suspend/hibernate mode. In case of an OS crash UEFI may try to make some kind of a rescue action.

2.3 UEFI Specifications and Vendors (IBVs)

UEFI forum is a joint effort driven by companies in the PC business. These companies include PC OEMs (such as IBM, HP and Dell), processor manufacturers (such as Intel and AMD) and Independent BIOS Vendors (IBVs such as AMI, Insyde and Phoenix). These companies have agreed upon UEFI, EDK and PI specifications. This sub-section
should give the reader an overview of the specifications and the solutions available in the industry for UEFI implementation.

2.3.1 UEFI/EDK Specifications
The UEFI forum has defined 5 sets of specifications: [14]

1. UEFI Specifications
2. UEFI Shell Specifications
3. UEFI PI Specifications
4. PI Distribution Package Specifications
5. ACPI Specifications

However, we would briefly look into UEFI Specification and UEFI PI Specification. Another important set of specifications are the EDK II specification. EDK II is a development code base for the implementation of UEFI maintained by Tianocore. So, for the basic implementation of UEFI the 3 specifications (UEFI Specifications, UEFI PI Specifications and EDK II Specifications) are important.

Version 2.4 of the UEFI Specification describes an interface between OS and the platform firmware. The specification contains implementation details and description of the following: [13]

1. Boot Manager: The task of the Boot Manager is to load UEFI drivers and applications in the order prescribed by global Non-Volatile Random Access Memory (NVRAM) variables.

2. EFI System Table: The name of this section may be a bit misleading. This section gives the implementation details of an entry point for UEFI image (UEFI Application, OS loader or a driver). EFI System table is just a sub-section, which points to the boot services table and runtime services table.
3. GUID Partition Table: This section explains the usage of GPT as a superior concept than the legacy MBR. It also explains how MBR is used in context of GPT and the benefits GPT has over MBR.

4. Services: The specification also explains the concept and implementation of UEFI Boot Services and Runtime Services.

5. Protocols: Multiple sections of the UEFI Specification define the protocols used within UEFI e.g. Media Access, ACPI, Network Protocols etc.

6. Security features: The protocols and functions required for implementing security features such as Secure Boot are specified. The purpose of Secure Boot and Driver Signing is to allow access to authentic resources.

7. Human Interface Infrastructure (HII): Finally the UEFI Specification specifies the protocol and functions to implement human interaction features such as, keyboards and mice.

Moving on to the UEFI PI Specifications, there are a set of 5 Specifications as of version 1.3. Following are the 5 different specification documents:

1. PEI Core Interface (Volume 1): This volume of the specification describes the implementation of the PEI phase. The main terminologies of PEI phase are explained. These include PEI Services table, PEI Foundation, PEIMs, PEIM to PEIM Interfaces (PPIs) and PEI to DXE Handoff. [15]

2. DXE Core Interface (Volume 2): This volume of the specification describes the implementation of the DXE phase. The document explains the implementation of DXE components i.e. Foundation, Dispatcher, Drivers, Architectural Protocols, Boot Services and Runtime Services Protocol. [16]

3. Shared Architectural Elements (Volume 3): This volume of the specification is targeted at Independent Hardware Vendors (IHVs) and firmware developers working with Firmware Volumes. Firmware Volumes are a logical firmware device. This document addresses the design and coding of HOBs and Firmware Storage. [17]
4. System Management Mode (SMM) Core Interface (Volume 4): SMM is the processor’s operating mode during System Management Interrupt (SMI), which is of high priority. This volume of the PI Specification describes the implementation and protocols relevant to SMM. [18]

5. Standards (Volume 5): This volume of the specification describes the implementation of the System Management Bus (SMBus) Host Controller Protocol and the SMBus PPI. [19]

Another very important set of specifications for the developers is EDK II which is the successor of EDK. EDK was specified and implemented by Intel or members of Intel Corporation and is now available at Tianocore. The implementation of UEFI is not just a C code but also require other metadata files. These metadata files include Module Information (INF) file, Package Declaration (DEC) file, Platform Description (DSC) file and Flash Description File (FDF). The EDK II has a specification for each such metadata file. Apart from these specification files EDK II has a Module Writer’s Guide, User Manual, Installation Guide and Build Specification file, which are very helpful to understand the development environment and workflow. Before looking into the specifications three terminologies must be understood. The first terminology is ‘Module’; it is an atomic piece of code that can be built separately i.e. at least one C or binary file. The second terminology is ‘Package’; it typically consists of a group of modules. The third terminology is ‘Platform’; it is basically a package but with additional metadata files. [20] We will briefly look into the metadata files specifications in a top down manner:

1. FDF: Flash Description File is created if the developer wants to generate a flash output. It is normally a file within a Platform. The specification document explains the usage of FDF by the parsing tool. Additionally, FDF Specification specifies the usage of parameters required in the file. FDF defines the regions of binary images e.g. Flash Device (FD) and Firmware Volume (FV) regions. [21]

2. DSC: A Platform within EDK II must contain at least one Platform Description (DSC) file. The DSC specification explains the parameters required to complete this .dsc file, along with relevant examples. [22]
3. DEC: A Package within EDK II must contain at least one Package Declaration (DEC) file. The DEC specification explains the parameters required to complete this .dec file, along with relevant examples. [23]

4. INF: A Module within EDK II must contain at least one Module Information (INF) file. The INF specification explains the parameters required to complete this .inf file, along with relevant examples.

5. PCD: Platform Configuration Databases (PCDs) are tokens within DSC, DEC and INF files. The developers can make use of these tokens to alter the modules as per their liking without having to change the source code e.g. a “Hello world!” program can be run using a PCD token (Print ((CHAR16*)PcdGetPtr (PcdHelloWorldPrintString));) in a C file. This token can be changed in the .dec file to output a different string. PCDs can be reused from another module. The PCD specification defines PCD protocols and PCD PPI.

![Figure 2.3: EDK II specific build process][19]

As mentioned earlier UEFI development using EDK II does not only mean coding C source files but also writing the metadata files. These metadata files (INF, DEC, DSC and FDF) are parsed giving makefile information. These makefiles are then used along with the source files to generate binary files.
2.3.2 AMI
American Megatrends Incorporation (AMI) is one of the famous IBVs. This and the following sections focus on the UEFI IBVs. The purpose of this section is to introduce their solutions for UEFI implementation. Historically, AMIBIOS provided a solution for BIOS development. With the advent of UEFI, AMI developed Aptio which is now Aptio V, since a transition was needed from EDK to EDK II. Visual eBIOS (VeB) is the Integrated Development Environment (IDE) that comes along with AMI’s Aptio V package. Now in Aptio V package, VeB supports the EDK II environment and specifications. Aptio V has UEFI debugging solutions as well as other support utilities e.g. AMI Flash Utility, AMI Diag (for UEFI diagnosis), AMI BIOS Configuration Program. [24]

IBVs such as AMI, Insyde and Phoenix implement modules in addition to the ones already implemented by Intel (or other silicon manufacturers). These packages are then sold to the PC manufacturers (such as Fujitsu, Dell and HP) who further customize UEFI for their customers.

Figure 2.4: VeB showing AMI and Intel implementations
The figure above shows a UEFI implementation containing AMI and Intel components in VeB.

2.3.3 Insyde
Insyde is a Taiwan based IBV. Their product InsydeH2O (Hardware-2-Operating System) provides the UEFI/BIOS solution. H2OIDE is the Integrated Development Environment used by InsydeH2O. Apart from the IDE InsydeH2O has a diagnostic and debugging tool named H2ODDT, with source level debugging capability. [25] Out of the four debugging types allowed by EDK II, source level debugging is the best and most recommended. [9] Another utility that seems to be a good concept is EZH2O. It allows a user limited access to functionalities that do not require great BIOS knowledge. These functionalities can be modified or implemented in the absence of a BIOS expert e.g. modifying logos or splash screens. EZH2O may be useful for PC manufacturers who do not have their own BIOS team but want to customize the labels/logos under their own brand.

2.3.4 Phoenix
Phoenix Technologies was one of the first IBV to provide a solution for PC BIOS firmware, founded in 1979. Their UEFI solution comes with the name of Phoenix SecureCore. Phoenix released SecureCore Technology 3.0 on 28th of November 2012 transitioning from EDK to EDK II. [26] The previous SecureCore 2.x version claimed to have an improved architecture, from a monolithic one to a more modular one. This is one of the key benefits of UEFI. SecureCore also claimed to speed-up the boot time by parallelizing device initializations that were traditionally serially initialized. [27] One of the highlights of UEFI is its shift from assembly code to C (high level code). SecureCore also supports the assembly language code. Visual studio 2005 IDE can be used for SecureCore code base because it supports assembly code as well as C code. [28]

The dark side of Phoenix was its slower response to make the transition from BIOS to UEFI, compared to AMI. Phoenix had been leading the BIOS market but
this doesn’t seem to be true during the transition to UEFI. One reason to believe this is that Fujitsu Germany’s BIOS team, which was Phoenix’s customer, was not convinced with their solution for UEFI and shifted from Phoenix to AMI’s Aptio. Phoenix may have lost other such customers during the transition.

2.4 Lack of UEFI Testing

Software testing may have hundreds of definitions but one good and simple definition is “Testing is the process of executing a program with the intent of finding errors.” [29] Embedded systems software is not one of the easiest to test, debug and fix. One of the reasons is that an embedded system may not have a user interface at all to exactly figure out the problem just by booting the system. Sometimes the outputs of an embedded system may just be a collection of blinking Light Emitting Diodes (LEDs). Then again, it is not an easy task to figure out if and where the problem exists. Similarly the options for a black box testing within embedded systems compared to a computer application or a website may be very limited. There might only be a handful of user input methods/buttons for testers to show their art.

Even before these testing issues come into play, a prerequisite for embedded systems is the correct functioning of hardware platforms. This in itself is a humongous task. Moreover, even if the fault is found during the testing of an embedded system, there is always a margin of uncertainty. The tester, nor the developer, can be immediately convinced whether the issue arose because of a hardware problem or a software problem. Yet again embedded code may work for some hardware architectures and not for others. Another important term in the field of computer science is “Real time” systems. This term too has a great impact on the embedded systems. For a real time system it must be ensured that the result of a computation or processing arrives in time i.e. before it is too late and the result becomes useless. This makes testing of embedded applications even more challenging. The value of embedded systems is undeniable; this leads the field into ever growing scrutiny. In the recent past industry standards (such as EN, ISO and IEC) have become binding in some areas of embedded system applications. Similarly the tasks of testers have become more difficult. Not only do they have to test the functionality of a system, neither only the timing constraints but also
ensure the system complies with the industry standards. One of the characteristics of embedded systems is its resource constraints. Thus, a tester or the test tool may not have the liberty to consume a huge amount of space on the target platform. [30]

Until now we have discussed the technical aspects that make embedded software testing challenging. However, the misery does not end here. As per the Software Development Life Cycle (SDLC) testing is done at the very end. Even if an iterative SDLC process is followed, testing stays at the end of the iteration. If the project is running late, it would be most common to see that the last step in the SDLC is done away with. As mentioned earlier this last step happens to be testing. As a result some disaster in the field may occur. Though things have changed over time and software testing is seen as a critical part in the industry, yet the focus is not as much as witnessed in other engineering fields. A building may not be open to general public until its architecture is approved and declared as safe for public usage. Likewise extensive tests are conducted before a mechanical product is released. Previously, certification authorities failed to realize that these mechanical, electrical, architectural products with seemingly a small amount of software had equally high potential of being fatal. [31]

The problems with testing UEFI are no different than the ones mentioned above. However, in addition UEFI has its own challenges to be taken care of. First of all it is a standard which replaces two and a half decades old solution (BIOS). It is not meant for a specific hardware platform or a specific OS. UEFI should be somewhat independent to both, the hardware platform and OS. Moreover, UEFI has to handle some complex security features. These security features need to be tested as well. UEFI is an open source project. The final code running on a consumer PC or server may contain code from multiple companies. This code base may be extremely large (approximately 18,000 files in a project that I’ve worked on). Apart from the source files numerous other types of metadata files are part of a final UEFI project. There aren’t many BIOS/UEFI developers in the industry compared to developers in other Computer Science fields. According to my supervisor “BIOS developers don’t grow on trees”. The testers are usually lesser in number than the developers. So we can safely assume that there aren’t many BIOS/UEFI testers either but the question is; are there any BIOS/UEFI testers at all? Maybe, but certainly not as many as needed in the industry. Here’s what Matthew Garrett thinks about this problem “UEFI is poorly tested in the
real world, UEFI contains a lot of code, UEFI contains an incredible number of bugs.” [32] Here’s what he says while giving a presentation on UEFI “For complex systems, Bug-Free does not exist!” [33]

Having discussed the importance of testing UEFI, it is a pity that UEFI code hasn’t gone through much of verification and validation. The literature on UEFI testing is not much and unfortunately the literature on testing of BIOS isn’t much either. If there would have been a focus on BIOS testing, these experiences could have been partially scaled to UEFI. Some of the reasons for this lack of UEFI testing are as follows: [33]

1. The code base is extremely large and code reviews are not very practical for such a large amount of code.

2. The configurations within UEFI are substantial and thus, a lot of testing scenarios are possible.

3. The industry is under a great pressure to develop UEFI solutions as quickly as possible to remain in the competition. This means that the companies want to get their product/solution out in the market as fast as possible, compromising on the testing required.

4. The tests have to be executed on all the platforms UEFI is expected to run on. This again could be a huge effort.

2.5 Issues with testing UEFI

Out of the little testing that UEFI gets, is not done in one of the most satisfactory ways. Currently the most popular ways in the industry to test UEFI are debugging, manual testing and diagnostics. We will look at these one by one.

A common misconception is to perceive debugging as testing. This perception is not true. Debugging is a developer’s task which involves locating the exact problem and fixing it. From an embedded systems perspective debugging in done using a tool and a device (debugger). The UEFI developers may debug the code in order to verify if the functions they have implemented work as expected. However, the main purpose of debugging is to fix an issue that has been reported. Debugging allows the developer to traverse through the code while looking at the output of the system. The point where the
error/issue occurs is noted and needs to be fixed. However, the developer should first know what to look for while debugging. It should be clear to the developer how the problem affects the functioning of the system. In order to know these two fundamental things a tester has to test the system first. Once a bug in the software or the system is found, the tester reports a bug and the procedure he followed to reach that state. Thus, testing is a prerequisite of debugging and they are not the same.

Secondly a lot of manual testing is currently being done in the industry. Though many experts may argue the advantages of manual testing over automated testing, one very famous disadvantage is that manual testing is very time consuming. This is particularly a problem in a fast growing field such as UEFI. The lack of time forces the companies to release their product without thoroughly testing it. Apart from the timing issues UEFI code partially runs before there are any resources or OS available. Manual testing under such circumstances may leave a lot of gaps. Hence, the manual testing can test basic features such as the following: [34]

1. Upgrading of UEFI versions.
2. Correct functioning of options from the setup menu.
3. Correct traversing of POST codes.
4. Testing LEDs and displays.
5. Changing of the boot orders.
6. Running different OS on UEFI.
7. Sending the system into sleep states and waking them up.
8. System trip points.

These tests are partially testing some of the phases in UEFI. However, other phases such as SEC and PEI phase may remain completely untested.

Lastly, there are few diagnostic tools available from different PC manufacturers. They perform automated hardware tests. Following are a few diagnostic test utilities:
1. **AMIDiag**: This utility by AMI performs hardware diagnostics. It tests system components such as ports, processor and memory tests.

2. **ePSA**: *Enhanced Pre-Boot System Assessment* by Dell comes along with Dell computers and can be run from the boot menu. It performs hardware test such as video graphic, battery, CD-ROM, ports, processor and memory tests.

3. **IBM DSA**: *IBM Dynamic System Analysis* is a Diagnostics tool for the IBM PCs. It performs system hardware diagnostics as well.

4. **MemTest86**: This is a test utility by PassMark software that performs memory tests for x86 architectures. The idea behind this test is to see if writing data to a cell would affect the neighboring cells. In practice MemTest writes zero to a cell, one to the neighboring cells and checks whether the first cell still has a zero. [35]

There are a number of other similar diagnostic utilities. However, this is not what we want. These are all hardware diagnostics whereas our goal is to improve the UEFI code quality. Running these diagnostic tools will mostly pass all the tests, hence not pointing out serious quality gaps. Unfortunately if we look for the UEFI/BIOS testing tools/utilities on the internet, we will find a lot of these hardware diagnostic utilities. These diagnostic tools may be good for the customers as an acceptance test but the testers of UEFI would want something more than this.

### 2.6 Summary
To sum up the state of the art, it was evident that BIOS had become unmanageable and the industry required a new standard. This led Intel to take an initiative and they came up with EFI. Learning from the mistakes of BIOS Intel decided to donate their work to the industry. EFI then became UEFI and the UEFI forum drafted specifications to formalize and standardize UEFI. The work is still on going and the industry makes use of EDK for the development of UEFI. IBVs have shifted their focus from BIOS to UEFI. Hence the development of UEFI is going on at a very high pace. However, the quality assurance of UEFI is not catching up with this high speed development. Having stressed upon the importance of testing UEFI, there seems to be a little headway in this area. The testing of UEFI is at least as important as in any other embedded systems firmware fields. Yet most of the assumed testing is either debugging, manual or...
hardware oriented. Debugging is meant to fix the bugs and technically, different to testing i.e. finding the bugs. Manual testing on the other hand may be too time consuming and weak in terms of UEFI. Finally the diagnostic tools tend to point out hardware issues whereas; the software issues maybe more obscure and a target for quality improvement. Hence a need for automated UEFI testing is clear.
Chapter 3

3. Available UEFI Test Tools

The first and foremost technique is to look for already available tools and avoid reinventing the wheel. In this chapter we take a look at the available tools and see how they can help us improve our UEFI code. *Firmware test suite (fwts)* is a test tool that comes as a package within Ubuntu. *Platform Initialization Self-Certification Test (PI SCT)* is a test tool that verifies the implementation of PI specific functions. It is available at UEFI Forum. *Chipsec* is an open source platform security assessment framework by Intel. In the course of this chapter we should understand the purpose and functioning of each of these test tools. Before moving into the details of each of these tools, another test tool is worth mentioning; *Linux UEFI Validation (LUV)*. Currently the 4th version of LUV is available. LUV uses *fwts* and aims to test UEFI with respect to Linux OS. However, it does not execute all the possible tests within *fwts* and the results log is not as reader friendly as that of *fwts*. Nevertheless, LUV is worth keeping an eye on as the project progresses. LUV will not be discussed in more detail in this thesis as it would be a duplicate of *fwts*.

![Diagram of UEFI phases](image)

*Figure 3.1: Complete coverage of UEFI phases using the test tools*
3.1 Firmware Test Suite

Firmware Test Suite (fwts) is a firmware testing tool by Canonical that aims to find bugs in the UEFI/BIOS firmware. It is an automated test tool that has 2 versions; fwts and fwts-live. fwts requires Ubuntu OS which allows fwts as a .deb package to be downloaded and executed. On the other hand fwts-live can be booted from an external device such as a Compact Disk (CD) or a Universal Serial Bus (USB). fwts tests UEFI from an OS perspective, in the process it catches Linux kernel warnings, suggests possible resolution and states variable names used in the UEFI code, making it easier for the developer to debug the code. The test tool supports all features for the x86 architecture but just limited features on the ARM architecture. [36]

fwts consists of 6 classes of tests: [37]

1. Batch: These are the basic BIOS/UEFI oriented tests and of great use for improving UEFI code quality. This class of test consists of approximately 45 tests. Notable tests among these 45 tests are:

   a. ACPI tests: Advanced Configuration and Power Interface (ACPI) provides an interface for Power Management functions to be used by the OS. Hence the ACPI tables and other related features are checked which play a very important role in the last phase of UEFI (After Life) when the system goes in different power modes. The syntax for ACPI machine language is also checked by disassembling and assembling the code and comparing it with Intel IASL assembler.

   b. BIOS tests: These tests check whether the BIOS 32 Service Directory standards are met and remain usable by the kernel. Additionally the tests related to BIOS support, security and tables are ran.

   c. Processor tests: These tests check if the CPU related states (P-states, C-states) are consistent for all CPUs and make sense e.g. higher performance states actually perform higher.

   d. DMTF tests: Distributed Management Task Force (DMTF) is a working group that has standardized management of IT infrastructure. Desktop Management
Interface (DMI) and System Management BIOS (SMBIOS) are two of their management standards. DMI is an interface that manages the components of a computer system. SMBIOS standardizes the way computer vendors present their management information. [38] The tables of DMI and SMBIOS are checked for warnings, errors and bad initializations.

e. Kernel tests: The kernel logs are read for system information, warnings, errors and oops messages. Oops messages are treated as critical.

f. UEFI tests: The tests for UEFI include information about the presence of Compatibility Support Module (CSM), secure boot variables and secure boot certificates. UEFI run time tests are performed. These tests ensure the availability of run time variables during the RT phase of the UEFI.

2. Batch Experimental: This test class seems to be a place where new unstable tests can be added and ran. As the name suggests these are beta quality tests and can be experimented with. However, running this class may not run any test if there is no beta quality or experimental test.

3. Interactive: This class of test requires input from the user. As the name suggests the user interaction is required during this testing. The user is required to press the power button to see if the ACPI functionality is implemented for this task. Similarly the brightness test tries to dim and brighten the monitor and asks the user if the background light changed. The ac-adapter and battery power tests require the user to unplug and plug the power cable. The keys are pressed to figure out that the key map file is correctly configured. Lastly, the lid test figures out if an event occurs when the laptop lid is closed. The problem with these tests is that the test class assumes the computer system to be a notebook or a laptop at times, which may not be true.

4. Interactive Experimental: Like the Batch Experimental test class, Interactive Experimental too seems to be a place where new unstable tests can be added and ran. As the name suggests these are beta quality tests and can be experimented with. However, running this class may not run any test if there is no beta quality or experimental test.

5. Power Management: This test class is related to the power states; s3 and s4. The s3 tests the suspend/resume implementation at the firmware level along with the amount of
power consumed in the process. The s4 tests hibernate/resume implementation at the firmware level.

6. Utilities: This class does not test anything rather it is more of an information gathering class. The class could be run to gather dump information. This information can be used by the developers to debug the firmware issues. The dump information for ACPI, Complementary Metal-Oxide Semiconductor (CMOS) memory, Extended BIOS Data Area (EBDA), system memory map, multi-processor tables, ROM and UEFI are logged.

The result log file shows information regarding each test that was ran. A very useful feature of fwts is the advice that is given after some failures, which makes it easy for the developer to debug and fix the issue. Another key feature of fwts is the classification of failures. At the end of the result log a summary shows tests that were critical, high, medium and low. A neat table is produced which shows the number of Pass, Fail, Abort, Warning, Skip and Info corresponding to each test (Appendix A shows a result log).

Figure 3.2: fwts live [36]
The live version of *fwts* allows the tester to run the batch tests only. All batch tests can be run at once or individual tests can be selected. If testers require running other classes of tests they should consider using the non-live version under Ubuntu OS.

### 3.2 Platform Initialization Self-Certification Test

*Platform Initialization Self-Certification Test (PI SCT)* is an open source project that verifies the UEFI code with respect to PI specifications. PI specification consists of 5 volumes (as mentioned in section 2.3.1). The most important amongst these volumes are PEI and DXE. The specifications describe the functions and their usage during these phases of PI within boot process.

The PI specifications are a set of long documents and thus the SCT document, that describes the tests, is a long one too. I will start off with an example for one function mentioned in the PEI volume. This example could be scaled and should help the reader to understand other tests performed by *PI SCT*. According to the PEI volume of PI specification, the function `ResetSystem()` resets the platform. The status code that this function can return in case of a failure is “EFI_NOT_AVAILABLE_YET”. This status code means that the service has not yet been installed. [15] The *PI SCT* has 9 different tests to verify `ResetSystem()`. The function is tested against two variables; Reset Type and Task Priority Level (TPL). For this particular function both the variables (Reset Type and TPL) have 3 options each. TPL could be TPL_APPLICATION, TPL_CALLBACK or TPL_NOTIFY. The Reset Type could be ResetCold, ResetWarm or ResetShutdown. Hence the `ResetSystem()` should be tested against all possible combinations of these two variables. The SCT case specification states that *PI SCT* verifies all these 9 combinations. The system should be reset (or shutdown in case of EfiResetShutdown) when the function is called, in each of the following situations: [39]

1. EfiResetCold and EFI_TPL_APPLICATION
2. EfiResetCold and EFI_TPL_CALLBACK
3. EfiResetCold and EFI_TPL_NOTIFY
4. EfiResetWarm and EFI_TPL_APPLICATION
5. EfiResetWarm and EFI_TPL_CALLBACK

6. EfiResetWarm and EFI_TPL_NOTIFY

7. EfiResetShutdown and EFI_TPL_APPLICATION

8. EfiResetShutdown and EFI_TPL_CALLBACK

9. EfiResetShutdown and EFI_TPL_NOTIFY

Each of these 9 tests has a unique GUID. Likewise the unique GUID of all possible combinations of tests are listed in the “GuidFile.txt” along with their short description. This file comes with the PI SCT package. Not all but most of the testable functions mentioned in the PI specification are tested by PI SCT. Following are the Services/Protocols/Network Protocols that are tested by PI SCT: [39]

1. Boot Services: CreateEvent(), FreePool(), LocateProtocol(), StartImage(), Exit() etc.

2. Runtime Services Test: SetVariable(), GetTime(), ResetSystem() etc.

3. EFI Loaded Image Test: EFI_LOADED_IMAGE Protocol Test

4. Device Path Protocol Test: Device Path Node Conformance Test etc.

5. EFI Driver Model Test: GetDriver(), ForceDefaults(), RunDiagnostic(), Query() etc.

6. Console Support Test: ReadKeyStroke(), OutputString(), GetState(), Write() etc.

7. Bootable Image Support Test: OpenVolume(), Flush(), ReadDisk(), SendData() etc.

8. PCI Bus Support Test: PollIo(), MemRead(), Pci.Write(), Map(), FreeBuffer() etc.

9. USB Support Test: GetCapability(), UsbBulkTransfer(), UsbPortReset() etc.
10. **SCSI Bus Support Test:** ExecuteScsiCommand(), ResetChannel(), PassThru() etc.

11. **ISCSI Boot Test:** Get(), Set()

12. **SNP, PXE and BISTest:** StationAddress(), Dhcp(), Discover(), SetIpFilter() etc.

13. **Debugger Support Test:** GetMaximumProcessorIndex(), Poll() etc.

14. **Compression Test:** GetInfo(), Decompress()

15. **ACPI Test:** InstallAcpiTable(), UninstallAcpiTable()

16. **Managed Network:** GetModeData(), McastIpToMac(), CreateChild() etc.

17. **EFI Byte Code Virtual Machine Test:** CreateThunk(), GetVersion() etc.

18. **ARP and DHCP:** Configure(), DestroyChild(), RenewRebind(), InfoRequest() etc.

19. **TCP, IP and Configuration:** Connect(), Groups(), GetData(), ProcessExt() etc.

20. **UDP and MTFTP:** Cancel(), ParseOptions(), ReadDirectory() etc.

21. **VLAN and EAP:** Set(), Find(), Remove()

22. **EFI Tape IO to Test:** TapeRewind(), TapeSpace(), TapeWriteFM() etc.

23. **Security Test:** GetHashSize(), Hash(), Get(), Set()

24. **EFI Firmware Management Test Case:** GetImage(), SetImage(), CheckImage() etc.

25. **HII Test:** GetFontInfo(), GetSecondaryLanguages(), DrawImage() etc.
The PI SCT project still misses some PEI and DXE phase tests so the project is under continuous development. Since this is an open source project, the community can contribute to it. The PI SCT package contains UEFI Self-Certification Test Case Writer’s Guide which could be referred to while developing additional tests.

The PI SCT contains a prebuild binary that may be run from the UEFI shell on the platform to be tested. However, if the tester wants to build the project, that is also possible. The source code is available and the build process is described in the SCT Getting Started document. However, the build process is not very simple. To build the PI SCT project we require third party tools/libraries such as Tcl-Tk, Libnet and Winpcap. Furthermore, EDKII, EFI shell, Microsoft Windows, Microsoft Visual Studio (VS) and Windows Driver Development Kit (DDK) is required. The SCT Getting Started document mentions specific versions of VS and DDK; however, the commands for running the build process under different versions is given in the tools_def document of the PI SCT package. Sometimes the batch files would need to be adjusted according to the environment or the files would need to be copied into the right directories, hence the build process may not be as straightforward as it seems.

PI SCT consists of two modes; native mode and passive mode. The native mode allows the tester to use command line or menu-driven interface to run the PI SCT from the UEFI shell. The passive mode makes use of UEFI Management Side (EMS) and runs the PI SCT agent in the passive mode. Most of the network protocols can only be tested in this passive mode. The project can be run from an external drive such as a USB or CD however, my experience is that running PI SCT from the hard disk is more stable and consumes less time to be completed. Running the platform specific install binary file (.efi) copies and prepares the PI SCT. Once PI SCT is ready to run, the tester may either execute the command “SCT –a” to run all tests or “SCT –u” to enter the menu-driven interface. The tester may run specific tests if required. After the tests have been completed a report can be generated as a .csv spreadsheet. The overall test results are generated and saved in a log file. The log file shows one of the following results; Pass, Warning or Failure. The warnings maybe given due to the skipping of a test. A failure occurs when the output of a test case is not as expected. The test report shows the number of tests failed or passed within a service/protocol/network protocol. The details of the failed tests are listed first and then the details of the passed tests.
The PI SCT menu-driven interface allows the testers to select tests they wish to run, configure the environment or generate a test report (Appendix B shows a result summary).

3.3 Chipsec

Chipsec is another useful open source tool for analyzing UEFI security features and configurations. It is a platform security assessment framework developed by Intel which is available for the open source community. Chipsec can run on Windows, Linux or UEFI shell. The framework is developed in Python and allows developers to write and test other security features. Chipsec can be started by running the Chipsec executable from a Windows command prompt, Linux terminal or UEFI shell, after configuring the environment for each.
The core components of Chipsec are as follows: [42]

1. chipsec_main.py: The main application.

2. chipsec_util.py: The utility to access hardware resources.

3. chipsec/chipset.py: It detects the chipset.

4. chipsec/logger.py: It is responsible for logging data.

5. chipsec/file.py: It is responsible for reading and writing files.

6. chipsec/module_common.py: It is an include file for common modules.

7. chipsec/helper/oshelper.py: It is used to abstract OS specific kernel driver.

8. chipsec/helper/xmlout.py: It is an xml support for JUnit.

Apart from the core components Chipsec has components for Hardware Abstraction Layer (HAL), OS specific helpers and platform specific configuration. The source code is available and can be extended to write one’s own security modules as well. The Chipsec utility (chipsec_util.py) should be used with great care as it provides direct access to the hardware resources. Chipsec utility can be used to access SPI flash content, PCI configuration, UEFI variables, I/O Ports, Model Specific Registers (MSR), OS Interrupt Descriptor Table (IDT) and Global Descriptor Table (GDT).
The result log shows the result of each test ran, commenting on whether they were found to be secure or not. In case of a failure the result log mentions the reason and possibly a resolution (Appendix C shows a result log). The tests include checking:

1. Pre-boot passwords in the BIOS keyboard buffer are secure.
2. BIOS interface and top swap mode is locked.
3. BIOS region is write protected.
4. Compatible SMM memory (SMRAM) is protected.
5. CPU SMM cache poisoning / SMM range registers (SMRR).
6. SPI flash controller configuration is locked.
7. Protection of secure boot key and configuration EFI variables.
8. Attributes of secure boot EFI variables.

### 3.4 Summary
The three tools mentioned above are all available to the open source community. The work on these tools is still on going. Thus the tools are iteratively being improved. All these tools seem to be very specific and useful for UEFI.

To ensure a good coverage of UEFI features and phases, all these tools should be used in parallel. They have a different purpose and cover different areas hence the tools should not be considered as complementary to each other. The purpose of *fwts* is to figure out how well UEFI performs from an OS perspective. From a technical point of view it can be said that *fwts* tests/verifies the right half of the UEFI phase i.e. partially the TSL phase, the RT phase and the AL phase. On the other hand the purpose of *PI SCT* is to test/verify that the implementation complies with the PI specification. This basically targets the PEI phase, DXE phase and directly or indirectly the BDS phase. The *Chipsec* completes the UEFI coverage by ensuring that the security features are implemented. The SEC phase is partially covered by *Chipsec* and partially by *fwts* during system restarts.
Chapter 4

4. Test Driven Development for UEFI

The concept behind Test Driven Development (TDD) is to start testing during the code development stage. In the conventional development process, the developers have to implement a code which goes through the testing process and lands back at the developer’s table, after the bugs have been found. The developer debugs and fixes the issue. This code goes through testing process again and lands back at the developer’s table, this time due to the side effects (bugs) of the previous fix. This time around the developer debugs and fixes the bug as well as looks for the side effects and the previous fix. Yet this process keeps on repeating for one issue or the other. TDD aims to change this process, or rather minimize the effort within this process. The developers write the tests first, which will obviously fail. Then they would write the code to make that test case pass. According to Edsger Dijkstra “Those who want really reliable software will discover that they must find means of avoiding the majority of bugs to start with, and as a result, the programming process will become cheaper. If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.” [43]

TDD is one of the less implemented software development processes in the industry. One of the reasons for this unpopularity is that it seems to be a time consuming development process. Moreover, there is not much literature discussing this topic. Amongst the literatures which do talk about TDD do not explain well the way to practically realize this process. Furthermore, as the testing of embedded systems isn’t easy, the test driven development isn’t any easier either. Amongst the reasons for this difficulty are the facts that hardware is also being developed parallel to the software, the hardware may have their own restrictions, the memory and hardware resources are limited, the hardware architecture may not be very simple and the hardware may have bugs too. The goal of my research in this area was to show how TDD can be realized for the implementation of UEFI.
4.1 Importance of Test Driven Development

As mentioned earlier TDD may seem to be a time consuming process but the fact is that implementing TDD should decrease the overall development time. Theoretically it should save the debug time at the end of the SDLC (bug fixes). Though the coding and testing time may increase but the debug time should decrease more, resulting in an overall decrease in development time. Secondly, we want to start the testing as soon as possible in a SDLC. This would help us having some confidence in the product even if the testing has to be somewhat compromised due to the shortage of time. Some of the bugs in the code may not be detected by a human eye while conducting a code review. For this reason it is important to run the code for specific values where the code could possibly fail. Using this test first approach the developers would tend to write a simpler code which is easier to test. Furthermore, the test coverage using TDD should be greater than the test last approach. Ideally all the code written has been tested for at least some scenarios, if not all. Using the conventional development methods, the test cases are written considering the software requirements and so is the code. This means that the code and the tests have an indirect relationship. In TDD there is a direct relationship between the tests and the code. In parallel to the production code, the developer is also developing an automated test suite. These test suites can be used as regression tests later when the code changes.

![Figure 4.1: Debug Later Programming](image-url)

Figure 4.1: Debug Later Programming [44]
In the conventional way of programming, there is a huge time between a coding mistake and its discovery because the testing is done long after the code has been developed. Bug found refers to the debugging of an issue and finding the root cause of the problem. This too takes time as the tester first reports the problem which the developer debugs. The root cause may be covered in a heap of code and hence not easy to get to. Depending on the type of bug the fix or work around may require from a very small amount of time to a medium amount of time.

For TDD the cycle starts with making a mistake. The developer writes a set of tests, runs the code and the tests fail (as there is no production code to pass these tests). Now the programmer writes the code for this particular set of tests. All the tests should now pass. This is a very small cycle compared to the one in which conventional programming is done. However, there are numerous such cycles in a TDD, ideally for every function implemented. In the process, the developer can better understand the problem being solved. The developer has a clear idea of what the results should be. Once the tests pass, developer has more confidence on the code. TDD ensures that developer has minimum debugging to do at the end of the software life cycle. This means that the developer would have less chance of introducing new bugs while fixing old bugs. The tests are also a source of documentation. They tell which scenarios have been tested. Implementing TDD helps the developer monitor the progress. This progress is not only an implementation progress but a more reliable development progress i.e. the code is not only written but also working for different test scenarios. [44]
To statistically compare a conventional SDLC with a TDD is difficult. There are a number of variables that may affect the results. There are fewer bugs if the focus on testing is increased but to prove that the design becomes better, or the time of the development cycle decreases significantly is not so easy. [45]

4.2 Unit Tests
Let’s look at how we can practically visualize the implementation of TDD. Unit tests are source code files that contain compare functions. These compare functions help the developer to program the test codes. There are several unit test harnesses available. Following are a few unit test frameworks that can be used for TDD in C:

1. Unity
2. CppUtest
3. Opmock
4. AceUnit
5. Check
6. Cgreen
7. Cmockrey
8. CuTest
9. CUnit
10. CUnitWin32
11. CUT
12. Cutter
13. EmbUnit
14. MinUnit
15. Seatest
Most of these unit tests are open source and based on xUnit. Similarly some of these can be used for embedded systems that have their implementation in C language, UEFI for example.

Since I used Unity for my research purposes, I will introduce this unit test framework. Unity is a lightweight unit test framework developed in C. By lightweight we are talking about a source file along with two header files. Unity can be implemented for embedded systems. However, the Unity project allows the developer to take their implementation of TDD to the next level. Thus, the developer may choose to go for Cmock or Ceedling. Cmock is a tool that uses Unity and mocks the interfaces. It does this for each function in the header file making the work of the programmer easier. This means that for each function, other functions will be automatically generated, serving as comparisons. Cmock is scripted in Ruby. [46] Ceedling makes things even easier. It is a build system for C language projects. If developers don’t want to configure the system manually or want to get rid of the makefiles but are ok with scripting in Ruby then they should consider using Cmock or Ceedling. On the other hand if the developers want to work in C language only and test separate modules, Unity maybe a better idea.

All tests or group of tests are run in the main function. Before each test case/function runs, test setup function is executed. The setup function makes sure that the environment is fit for the test case to run correctly. After each test case/function runs, test tear down function is executed. The tear down function cleans up the changes in the environment caused by running a test. It ensures that the following test runs in the correct environment. Following are some of the test conditions that are available in Unity: [44]

1. **TEST_ASSERT_TRUE** (BOOLEAN CONDITION): This is a check to verify that a Boolean condition is true.

2. **TEST_ASSERT_FALSE** (BOOLEAN CONDITION): This is a check to verify that a Boolean condition is false.

3. **TEST_ASSERT_EQUAL_INT** (EXPECTED, ACTUAL): This condition verifies that the number given in EXPECTED is equal to the value contained in the variable ACTUAL.
4. **TEST_ASSERT_EQUAL_STRING** (EXPECTED, ACTUAL): This condition verifies that the string given in EXPECTED is equal to the string contained in the variable ACTUAL.

5. **TEST_ASSERT_POINTERS_EQUAL** (EXPECTED, ACTUAL): This condition verifies that the pointer given in EXPECTED is equal to the pointer ACTUAL.

6. **TEST_ASSERT_FLOAT_WITHIN** (EXPECTED, ACTUAL, TOLERANCE): This condition verifies that the floating number given in EXPECTED is equal to the floating value contained in the variable ACTUAL ± TOLERANCE.

7. **TEST_FAIL_MESSAGE** (MESSAGE): The test is shown as failed, stating the reason in the message.

Usually a hard coded value goes into EXPECTED and a variable or a function in the ACTUAL parameter. There are several other comparisons/assertions/checks available in the Unity header files that can be used for TDD.

### 4.3 Test Driven Development in Aptio V

During my research I performed TDD in AMI’s Aptio V Integrated Development Environment on UEFI code. Before starting to prepare Aptio V for TDD it was necessary to understand the working of Aptio V. With the coming of EDK II the metadata files such as DEC and INF, had to be implemented. Moreover Aptio V has files specific to the IDE such as .cif and .sdl. Finally the C (source code) files have to be coded.

To better understand the structure and implementation of UEFI modules in Aptio V I started off by writing a template package. This template package includes files and code to implement functions in the DXE phase, the PEI phase and a UEFI application that can be run from the UEFI shell. Moreover a library template for each afore mentioned functionalities were implemented. The libraries are helpful as they can be used by other modules.
A good way to start an implementation of a UEFI package is to create a separate folder and subfolders on your system and create all metadata files one by one. Once all the metadata files have been created, the whole directory can be added as a component in Aptio V project. Each package (technically eModule) such as GE_TemplatePkg should be a separate folder. This folder should contain a .cif, .dec and .sdl (AMI Aptio V specific) files. The folder may also contain subfolders for each functionality (technically ModulePart) such as GE_TemplateDxe, GE_TemplatePei, GE_TemplateUefiApp and GE_TemplateLib. Each subfolder, implementing a functionality, should contain a .cif, .inf, .sdl and source code/header (C) files. A .cif file contains the architecture of the package/component/module e.g. GE_TemplatePkg.cif states that the GE_TemplatePkg.dec, GE_TemplatePkg.sdl and GE_TemplateDxe, GE_TemplatePei, GE_TemplateUefiApp, GE_TemplateLib components are part of this package. A .dec (Package Declaration) file contains data as per the EDKII specification. The .sdl file is AMI Aptio V specific and is not specified by EDKII. SDL stands for System Description Language and describes system data for parsing and code generation. [47] The .inf (Module Information) file contains data as per the EDKII specification. The
INF components (specified by the .inf file) contain the source code files and header files. Once the folder is prepared with completed metadata and code files, it can be added to the Aptio V project as a component. The developer has an option to either add the whole directory or the .cif file. The .cif file will automatically include all files/components written in it. To build the template, C files should have at least an entry point specific to the Module Type. This entry point can be assumed as similar to the main function for standard C code development.

The TDD was implemented for 3 Module Types; UEFI_APPLICATION, PEIM and DXE_DRIVER. UEFI Applications are those applications which can run from UEFI shell and are unloaded when they exit. PEIMs are the PEI Modules used during the PEI phase. DXE_DRIVER are modules which execute in boot services environment. The implementation of each of these 3 Module Types contained 2 Module Parts. One Module Part consisted of Unity project, while the other Module Part consisted of an example source code.

![Image of TDD Package structure in Aptio V](image)

Figure 4.4: TDD Package structure in Aptio V

The file structure for all three implementations was the same. However, the Unity code couldn’t be used as it was. This is because the original Unity code used the standard C libraries such as `<stdio.h>`, `<setjmp.h>`, `<limits.h>`, `<stdint.h>` and `<string.h>` that are not available in the UEFI (EDKII) specific code. Nevertheless, Unity has kept standard C specific code as little as possible so that it can be easily modified for embedded systems that do not have standard C libraries.
The first task was to implement the `putchar()` function as it is defined in the standard C library but not in the UEFI code. The function outputs one character at a time. Other modifications included fixing `UNITY_PRINT_EOL` (End Of Line) for aesthetic purposes. The function `UNITY_PRINT_EOL` was only using the newline (`\n`) character and so the output started on the next line but not from the left most position. Thus carriage return (`\r`) was added in this function. This ensured that the output which was meant to start from the next line, was started from the next line and from the left most cursor position rather than where the cursor was left from the previous line. The functions `TEST_PROTECT()` and `TEST_ABORT()` were removed from the Unity code because they consumed the standard C library (`<setjmp.h>`). These functions abort an infinite loop caused by a test and so were not very important.

The source files in example Module Part contained 4 files; `ProductionCode.h`, `ProductionCode.c`, `TestProductionCode.c` and `TestProductionCode_Runner.c`. The first two C files (`ProductionCode.h` and `ProductionCode.c`) contain the code that will be part of the product. `TestProductionCode.c` contains the setup, teardown and test functions which test `ProductionCode.c`. The definitions of tests are available to `TestProductionCode.c` by including Unity header file. `TestProductionCode_Runner.c` contains the entry point (main). This source file is responsible for running all the tests. `unity.c`, `unity.h`, `unity_internals.h`, `TestProductionCode.c` and `TestProductionCode_Runner.c` are for the purpose of TDD and will be removed before release. Only the `ProductionCode.h` and `ProductionCode.c` will be part of the final release.

A binary file (.efi) will be generated for UEFI Application after the project is built. This binary file can be run from the UEFI shell. The tests listed in `TestProductionCode_Runner.c` will be displayed in UEFI shell. The developer can see which tests passed and which ones failed. The code for the failed tests should be implemented or modified to make sure that the tests pass. If all tests pass the developer should write the tests for the next feature. This is an iterative process.

For DXE_DRIVER and PEIM Module Types, we do not have the leverage of UEFI shell. The PEI and DXE phase run before the UEFI shell is available. Hence a different strategy should be devised in order to see the results of TDD. Files that are generated after building the DXE_DRIVER and PEIM Module Types are not UEFI shell
executable. A possible way to see the TDD results for PEI and DXE phase is to use the debugger. This wasn’t an easy task. Thousands of files were searched to find a function that would be suitable for sending the TDD results to debugger. SerialPortWrite() proved to be such a function. Our putchar() function (used for UEFI Application) had to be modified to write the output to the serial port rather than console output. Once this was done and the project was built successfully, the code had to be flashed to the target platform. The platform was attached to the debugger and turned on. The debug output contained the results of TDD.

The process of implementing TDD in UEFI (after the unit test files are imported and modified) is as follows:

1. Write a test function containing tests for many critical values (boundary values).
2. Include this test function in the test runner/entry point/main.
3. Build the code
4. Assuming that the code is built successfully, run the binary file in UEFI shell (for a UEFI Application) or burn the code to a target platform and turn on the platform to see the debug output (for PEI and DXE functionalities).
5. Verify that the newly written test function failed.
6. Write the code to implement the functionality and cover all test scenarios.
7. Perform step 3 and 4, verify that all tests pass.
8. Refactor the code i.e. clean the code as you would like to see in the final production.

4.4 Summary
This chapter explains the basic concept of Test Driven Development (TDD). It shows the importance of TDD as well as the challenges faced with respect to embedded systems. The difference between the working of a traditional development process and the TDD process is explained. The benefits of using TDD from a developer’s perspective are mentioned. It is also stated that statistically comparing the benefits of
TDD against the traditional development process is very difficult. This is because there are a number of variables involved.

The chapter further describes how to practically visualize the implementation of TDD. This is done by using unit tests. A number of open source unit tests for C language are listed. These unit tests would have to be modified in order to make them useable for embedded systems that do not contain standard C libraries.

Finally the TDD in AMI’s Aptio V for UEFI code is thoroughly explained. During the research, TDD was implemented for 3 Module Types; namely DXE_DRIVER, PEIM and UEFI_APPLICATION. The results of TDD can be seen in the UEFI shell for UEFI_APPLICATION. However, for the results of DXE_DRIVER and PEIM implementations, debugger has to be used because the console output is not available during the PEI and DXE phase. The chapter closes with a stepwise process to implement TDD in UEFI.
Chapter 5

5. Code Analysis for UEFI

Code analysis or source code analysis are techniques to analysis the code automatically. These techniques help the developers to fix or improve their code. At times the gaps in the code are not immediately evident. The code analysis tools can be used to point out these gaps. The manual code reviews are time consuming and may not be able to detect simple/hidden issues. I say this because the human eye and mind is looking for critical issues and might skip presumably small issues such as wrong bracket placements. Therefore, code analysis is a good alternative to manual code reviews, saving time and resources. Moreover, the developers of UEFI/BIOS are so scarce that they cannot be consumed in code reviews, all the time. The code analysis techniques could be divided in two; static code analysis and dynamic code analysis. My research was focused on how these code analysis techniques could be practically implemented for UEFI code (written in C programming language). For this purpose some of the code analysis tools were reviewed and will be discussed in this chapter.

5.1 Static Analysis

Static code analysis is described as analyzing the source code without executing it. There are several static analysis tools available in the market. Most of them are not free of charge but some are open source. The purpose of my research is not to look into every static analysis tool available. The purpose here is to see whether the static code analysis tools can help us improve the C language code, written specifically for UEFI. Many static analysis tools have a support for common IDEs such as Visual Studio. The research was performed to see if these tools are any useful for IDEs which are not directly supported such as the ones implementing UEFI code.

The static analysis tool vendors do not only focus on finding as many issues as possible but also ensure that the false alarms are minimum or negligible. This makes the task of static analysis tools very tricky. In context of UEFI, the task is even more difficult due to the fact that the standard C libraries in UEFI are not available. If the analyzer expects standard C specific implementations and raises a concern, then this would not be of
great help to a UEFI developer. The static analysis could be configured at times such that they are run every time the build runs. This way of developing software ensures that the bugs are minimized and caught at the earliest stage. There is obviously no arguing about the fact that building a code is not sufficient a proof for a code to work properly and without bugs. Successful build just shows that the code is syntactically correct. This background was necessary before mentioning that these static code analysis tools may not perform the same tasks as a compiler. Static code analysis tools have a different purpose from compilers as well as dynamic analysis tools. Static analysis is focused on 3 basic tasks; source code error detection, compliance with coding standard guidelines and software metrics which show the properties of different modules and their interactions. [48]

5.1.1 PVS-Studio

*PVS-Studio* is a static code analysis tool by OOO "Program Verification Systems". The analyzer can be used for C and C++ code. The tool is not open source and so the license needs to be purchased. However, a trial version with limited amount of clicks can be downloaded from the website. It has a plugin for *Visual Studio, Embarcadero RAD Studio* but also a stand-alone user friendly version. *PVS-Studio* has defined 4 units of diagnostics; general analysis (GA), optimizations (OP), 64-bit code problems (64) and OpenMP code problems (MP).

![PVS-Studio analysis under progress](image)

Figure 5.1: *PVS-Studio* analysis under progress
The stand-alone version proved to be very handy for UEFI code that was built in Visual eBios (AMI Aptio V’s IDE). I assume that the stand-alone version of PVS-Studio should be equally useful to analyze a source code that is built using any other IDE. The reason for this assumption is that PVS-Studio monitors the build process on the system and performs the analysis on the files that were successfully built. This technique seems to be IDE independent. PVS-Studio allows the developers to select the level of issues (1, 2 and 3) and the type of issues (GA, OP, 64 and MP) to be displayed. It also allows the developer to mark issues as false alarms which should suppress that specific issue and mark the suppression as comment in the source code file.

There are a number of issues that PVS-Studio detects. These issues are divided into the following 6 categories: [49]

1. Problems related to code analyzer
2. Diagnosis of 64-bit errors (64)
3. General Analysis (GA)
4. Diagnosis of performance warnings (OP)
5. Diagnosis of parallel errors (MP)
6. Customer Specific Requests

Following are some of the important issues that were detected on the UEFI project: [49]

1. Call of function ‘NameOfFunction’ with variable number of arguments, ‘PositionOfArgument’ argument has memsize type: The problem with having a memsize type is that if the code is transferred to a system with different architecture (32 bit, 64 bit) there may be some issues.

2. Dangerous magic number ‘SomeNumber’ used: This is a similar problem where the value of a variable may change due to a change of architecture (32 bit, 64 bit).
3. The code containing the collection of similar blocks. Check items X, Y, Z, ... in lines N1, N2, N3: A similar code is detected and may be a copy-paste problem.

4. Implicit type conversion to memsize type in an arithmetic expression: The interaction of different memsize type variables takes place and this could lead to type conversion error for different architecture (32 bit, 64 bit).

5. The 'VariableName' variable is assigned values twice successively. Perhaps this is a mistake: This error points out that a variable is written twice successively without being read in between thus, the first assignment does not make sense as it is overwritten without being used.

6. More than one sizeof() operator is used in one expression: A calculation may be performed using two different sizeof() resulting in an incorrect output.

7. Explicit conversion from 32-bit integer type to pointer type: This issue points out that an explicit 32-bit integer to pointer conversion took place. memsize types should be used to store a pointer in an integer variable.

8. A part of conditional expression is always true/false: This may be dangerous as the condition as a whole may either be always true or may not be required if it does not vary.

9. It’s probably better to assign value to 'VariableName' variable than to declare it anew: A local and global variable may have a same name and the developer may have intended to use the one with external scope, without a need for a new variable (local variable).

There are approximately 300 other error descriptions. In some cases PVS-Studio’s static analysis seems to be helpful but still generates a lot of false alarms, with respect to UEFI. There are no standard C libraries in the UEFI code. However, PVS-Studio shows errors such as “#include <BaseTsd.h>” should have been used instead of defining data types anew. BaseTsd.h is not available for the UEFI code by default, rather processorbind.h and base.h perform this task. Similarly, some of the issues shown as “A part of conditional expression is always true/false” were due to the variables defined in .sdl file that is specific to AMI’s Aptio V. The developer may change the value in that .sdl file later. This .sdl file specific data is
not in PVS-Studio’s knowledge. Moreover, the UEFI code that the industry works on or has developed so far is mostly x86 64-bit architecture specific. The static analysis shows issues which suggest how the code should be changed to be portable. So as long as UEFI is implemented on x86 64-bit architecture these issues are not a big deal. However, they would be very helpful for developers who would want to scale the code for other architectures.

5.1.2 Cppcheck

Cppcheck is an open source static analysis tool for C and C++. The ultimate goal of this tool is to detect as many bugs as possible without allowing false positives. Cppcheck has plugins for several IDEs such as Eclipse and Visual Studio. There is also a user friendly stand-alone version of Cppcheck. The major features of Cppcheck are as follows: [50]

1. Checking out of bounds
2. Checking code for each class
3. Checking exception safety
4. Checking memory leaks
5. Warning on usage of obsolete functions
6. Checking usage of Standard Template Library (STL)
7. Checking unused or uninitialized functions and variables

Technically Cppcheck performs the analysis by traversing through the token list. Tokens are elements in an expression/equation e.g. \( a=b+c*d \) has 7 tokens. A syntax tree for an expression/equation is used as a source of token gathering. The syntax tree is formed considering the token precedence. The symbol database has complete information regarding the symbols i.e. arithmetic operations, punctuations, functions etc. Before the checker is ran value flow analysis is performed. Value flow analysis basically computes all the results and intermediate results of functions. [51]
Cppcheck can be easily used for UEFI C source code. The developer can select the directory of the code. This directory is then automatically searched for and extracts the C source code. The source code is then analyzed and the results are loaded in parallel. This makes the execution of Cppcheck static analysis tool, IDE independent. Cppcheck claims that the reported errors are rarely false alarms however just like other static analysis tools many bugs may go undetected. Again it is worth mentioning that the static analysis tools such as Cppcheck do not look for syntax errors. This task is left to the compilers, therefore static analysis tools are usually ran after a successful build. [52]

Once the analysis has been performed, the user has the option to select the severity level of the results. There are 6 levels of severity in Cppcheck:

1. Error: These are the bugs that Cppcheck sees.
2. Warning: This level of severity suggests a safe programming approach that may prevent the bugs.
3. Style: These are more of a code refactoring alarm that may suggest getting rid of redundant code.
4. Performance: This level of severity suggests ways to make the code more efficient in terms of performance.
5. Portability: These are portability warning, pointing out that the code may work differently on different architectures/platforms.
6. Information: This level of severity is not really a bug they are just for the information. These messages tell which checks did not run and what command line option should be used to run them.

In the UEFI code Cppcheck pointed out the following errors; possible null pointer dereferences, uninitialized variables/struct members, index out of bounds and Memory/Resource leaks. The warnings included pointing to a lot of redundant code, ineffective statements, suspicious usage of sizeof, usage of scanf without field width limits and members uninitialized in the constructor. The style suggestions included suspicious conditions (using assignment and condition in a conditional statement), variables assigned a value but never used, the scope of
variables can be reduced, same expression on both sides of or (||) condition. Amongst the few portability warnings was returning an integer in a function with pointer return type is not portable. Most of the performance issues were regarding successively writing to a variable without using it in between. The information massages were mostly regarding the skipping checks and suggestions on how they could be run. Cppcheck seems to give minimal amount of false alarms when ran on the UEFI code, yet not negligible.

5.1.3 CppDepend

CppDepend is another static code analysis tool. There are 3 different licenses that can be bought; Developer Edition, Build Machine Edition and Full Edition. However, as a student I was given a license free of cost for my research purpose for which I am grateful. CppDepend has Add ins for Visual Studio but also a build monitor that allows analysis on different IDEs. The analyzing rules are divided into several different groups:

1. Code Quality

2. Code Quality Regression

3. Object Oriented Design

4. Design

5. Architecture and Layering

6. Best Practices

7. API Breaking Changes

8. Code Diff Summary

9. Dead Code

10. Visibility

11. Purity-Immutability-Side Effects

12. Naming Conventions
13. Source Files Organization

14. Statistics

![Image](image.png)

Figure 5.2: CppDepend query rules and groups

The most important analysis group is Code Quality. The rules in this group check some critical errors such as “Methods with too many parameters”, “Methods too complex” and “Types too big”. If a function has more than 8 parameters, “Methods with too many parameters” rule will be violated. If the cyclomatic complexity is greater than 20 and the nesting depth is greater than 5 then the rule “Methods too complex” will be violated. Cyclomatic complexity is a measure of linearly independent paths in a method/function that is calculated using a control flow graph. Nesting depth on the hand is a count of the number of loops within a loop. “Types too big” are violated when a source code file has greater than 500 lines of code. These critical issues point out the complexity of a code which can be potentially dangerous. Similarly this class of rules shows other complexity related issues. These are not as critical as the above stated three rules. They include methods to refactor (due to complexity), methods which are possibly poorly commented, types/files that have too many methods or have a poor cohesion.
Code Quality Regression group of rules are specific to code changes. On the first run they would give no errors but when the code has been modified and analyzed again they may point to a number of violations. The violations include “From now, all methods added or refactored should respect basic quality principles”. The basic quality principles query is defined as; lines of code in a method/function should be less than 31, cyclomatic complexity in a method/function should be less than 21, maximum nested loop should be less than 51, nesting depth should be less than 5, number of parameters should be less than 6, number of variables should be less than 9 and number of overloads should be less than 7. Code quality regression also has rules that make sure that the methods are not more complex or larger than the previous code.

The Object Oriented Design group checks rules such as depth of inheritance, constructor’s call to a virtual method, abstract classes with many parameters etc.

The Design group of rules contains 3 queries. A formula calculates whether the type/source code file has many responsibilities, if so the violation of this rule occurs. Another design violation is when a nested type is visible i.e. it is not generated by compiler nor is it private.

Architecture and Layering group of rules give violations related to cohesion, coupling and dependencies. The rules check namespace dependencies, project cohesion and instability.

Best Practices set of rules include a rule to ensure that the keyword goto is not used. The API Breaking Changes and Code Diff Summary groups are also specific to code modification and point out the issues occurred possibly due to code changes.

The Dead Code set of rules shows concern if a type, method or field is found to be potentially dead. Dead Code is that code/function/source code file that may never be executed.

The Visibility group of rules suggests which fields should be declared as private. The group of Purity-Immutability-Side Effects suggests that the structs should be immutable and a field should be assigned from within its parent hierarchy type.
The group Naming Conventions detect aesthetic violations such as the name of type, method or field is longer than 35 characters. Other violations include name of a type beginning with lower character, giving the same name to namespace and types, static fields should be prefixed with ‘s_’ etc.

The Source Files Organization set of rules define 4 rules; Avoid defining multiple types in a source file, Namespace name should correspond to file location, Types with source files stored in the same directory, should be declared in the same namespace and Types declared in the same namespace, should have their source files stored in the same directory.

Lastly, there are two information providing groups; Statistics and Trend Metrics. Statistics rules provide information/statistics on the most used types, methods and namespaces. The Trend Metrics tells the lines of code, number of source files, number of methods, average of nesting depth for methods, third party fields etc.

For UEFI CppDepend analysis can be performed using the build monitor. The UEFI project is built using the command line and including the CppDepend’s build monitor executable as an option. This allows the build monitor to gather the build information of UEFI code and perform analysis on it. The analysis tool seems to be a very heavy one, it took days to complete the analysis and the system consumed a lot of resources.

CppDepend uses the Clang parser to parse the code. There are a number of problems while running the code with respect to UEFI. First of all, the clang parser gives errors as it cannot interpret some statements in a makefile. Secondly, a large amount of errors were generated due to the inability of parsing C files within 10 minutes and hence, they were skipped after 10 minutes. This time could be decreased from the settings to significantly decrease the overall analysis time. The detailed report, after the completion of the analysis showed architectural, design and complexity issues. CppDepend showed that there are 800 potentially dead types/header files. With EDKII the developers can implement libraries as different modules. These libraries can be accessed by C source files without including the whole path. They have to be coded in the .inf file as well. This work
flow cannot be tracked by CppDepend and hence these libraries are treated as dead types/header files.

Figure 5.3: Dependency Graph

Figure 5.4: Dependency Matrix
Figure 5.5: Metrics

**CppDepend** generates 3 Graphs/Metrics. These graphs show an overview of the software architecture. The dependency graph shows the relationships/dependencies of a class/module with another. For each module the graph points to the modules being used as well as the modules being consumed. The module is shown as enlarged or compressed depending on the lines of code it possesses. Another way of viewing these dependencies is using the dependency matrix. This is a mirror image of modules on the x-axis and the modules on the y-axis. Each module shows the number of dependencies on other modules. For a large project this matrix is more readable. Lastly the metrics view shows these dependencies as square boxes. Each big box contains many small boxes that indicate that these modules are dependent.

### 5.1.4 Tools Comparison

There is already a comparison available between **PVS-Studio** and **Cppcheck**. Though this comparison was carried out by **PVS-Studio** itself, **Cppcheck** claims that this comparison is an honest one. [54] However, the analyses were performed on the source code of 3 games and did not draw any conclusion at the end. In each of the comparison **PVS-Studio** detected more issues than **Cppcheck**. This does not
mean that the *PVS-Studio* is better than *Cppcheck* because of a number of reasons. One of the major reasons is that the intersecting errors in each case were less than the number of errors detected by *Cppcheck* alone. This proves that *PVS-Studio* is not *Cppcheck* plus some other unique errors. Another reason is that almost every static analysis tool is prone to ring a false alarm. Thus, more errors detected do not necessarily mean that the static analysis tool was better. The errors detected may not really be an issue. Then again, different tools will give different results based on the project analyzed. *PVS-Studio* has uploaded many other articles on their website. The articles are very interesting and they give an impression that *PVS-Studio* puts in a lot of effort to help programmers understand key terminologies and methodologies.

The project I analyzed was a UEFI project implemented in C using EDKII. The project was same for all 3 static analysis tools. *PVS-Studio* and *CppDepend* are both proprietary software whereas *Cppcheck* is open source software. However, *PVS-Studio* has a trial version and I was given a *CppDepend* license free of cost for my research purpose. In economic terms this makes *Cppcheck* better than the other two.

UEFI requires build monitoring to accumulate the build data, unless the code is ported and set to run in *Visual Studio* or another well-known IDE that is supported by *CppDepend* and *PVS-Studio*. The IDE used for my research was *Visual eBIOS* as part of AMI’s Aptio V. I used the build monitor for both, *CppDepend* and *PVS-Studio*. *PVS-Studio* had a more user friendly way of monitoring the build than *CppDepend*. *PVS-Studio* automatically started the analysis after the build monitoring was completed however *CppDepend* required an explicit loading of the .build log. *Cppcheck* on the other hand does not require the developer to build the project. The developer can just open the directory of the project and *Cppcheck* will analyze all the source files within this project. This approach has 2 benefits over the build monitoring approach. The first benefit is that this approach is more users friendly and the developer does not require building the project in parallel. The second benefit is that *Cppcheck*, unlike *CppDepend* and *PVS-Studio*, will also check the files that were not built. The UEFI project may have several modules that may be inactivated or serve as templates, yet it is better to analyze them. With
respect to user friendly and completeness criteria \texttt{Cppcheck} seems to work better than \texttt{PVS-Studio} which seems to work better than \texttt{CppDepend}.

\texttt{CppDepend} consumes a lot of time and system resources whereas the other 2 are faster. \texttt{Cppcheck} seems to be lighter than \texttt{PVS-Studio} but that’s just from a user’s perspective. A stress test may be required to give a precise estimate on this. Nevertheless, \texttt{CppDepend} is by all means the heaviest amongst these 3. \texttt{PVS-Studio} requires more time than \texttt{Cppcheck} because it has to first look for the compiler invocations during the build (build monitoring) and then perform the analysis on those files. Thus, with respect to time and resources \texttt{Cppcheck} seems to be better than \texttt{PVS-Studio}.

Up till now, the 3 tools were comparable. Beyond this it needs to be mentioned that \texttt{CppDepend} performs a different kind of analysis and is not directly comparable to \texttt{PVS-Studio} and \texttt{Cppcheck}. \texttt{CppDepend} gives a more insight on the design, architecture and complexity of the software structure. The analysis does not point out the issues within the source code rather it focuses on the code complexity. \texttt{CppDepend} can be used by software architects to improve the architecture. However, for UEFI it does not seem very helpful as many files cannot be parsed. The ones that are parsed point to issues that may not be very helpful e.g. the code complexity is useful but a very user friendly light weight open source tool can be used for this (\textit{Cyclomatic Complexity Analyzer, ccm}). \texttt{CppDepend} is useful for small projects that are growing. For a project as big as UEFI it is practically too weak. Apart from the size, UEFI code is not written by one company. This fact makes it even more difficult for \texttt{CppDepend} to help the developers fix and clean their architecture.

Now we can compare the 2 tools with similar purpose; \texttt{PVS-Studio} and \texttt{Cppcheck}. Apart from the already discussed benefits of \texttt{Cppcheck} over \texttt{PVS-Studio}, the levels of errors shown in \texttt{Cppcheck} have more meaningful names. \texttt{PVS-Studio} has 3 levels of errors namely; 1, 2 and 3. \texttt{Cppcheck} has 6 levels of errors namely; errors, warnings, style warnings, portability warnings, performance warnings and information messages. The errors in \texttt{Cppcheck} could be seen as real issues they include; memory leak, resource leak, uninitialized variables and possible null pointer dereference. The rest are warnings, suggestions or information. There is a
different set of errors that both *PVS-Studio* and *Cppcheck* can point out. *PVS-Studio* issued approximately 12,500 errors whereas *Cppcheck* pointed out approximately 4000 errors i.e. 1/3rd that of *PVS-Studio*. On top of it *PVS-Studio* had analyzed lesser files than *Cppcheck*. *PVS-Studio* has approximately 300 error definitions and though it’s not documented for *Cppcheck* it seems that *Cppcheck* has lesser than these. However, the number of false alarms in *PVS-Studio* is higher for UEFI. *PVS-Studio* generates more errors and this means that the developers have to dedicate more time in figuring out which issues are false alarms and which issues are genuine.

As a result, *Cppcheck* should be used for UEFI in any case to improve the quality of UEFI source code. However, the decision to buy *PVS-Studio* depends on whether the development team would like to spend money to improve the code quality more than what *Cppcheck* can ensure. Usage of *PVS-Studio* will require more time to detect and fix the genuine issues. According to the budget other options can also be looked into, such as *Coverity, QA-C*, and *LDRA Testbed* etc. Moreover one static analysis tool may not be sufficient because every static analysis tool will give a new and a different issue hence improving the overall code quality. The power of static analysis should not be under estimated. According to Dave Revell: “The more I push code through static analysis, the more I’m amazed that the computer boots at all.” [55]

### 5.2 Dynamic Analysis

Another code analysis technique is dynamic code analysis. It is described as analyzing the source code while executing it. Dynamic analysis points out runtime issues. It has a different purpose or way of looking at bugs than static analysis. The static analysis tool can be compared with another static analysis tool and dynamic analysis tool can be compared with another dynamic analysis tool. On the other hand, the static analysis technique could be compared with dynamic analysis technique but a static analysis tool cannot be compared to a dynamic analysis tool due to the difference of technique. [56]

Dynamic analysis detects runtime errors such as memory leaks, code coverage and race conditions. The benefits of dynamic analysis technique include minimization or elimination of false positives and execution of analysis without the source code. False
positives are difficult to occur because the error detected is not a prediction it is an error that actually occurred during the execution of analysis. Secondly, dynamic analysis can be run on the application directly, without the need of source code. At times source code may not be available as it can be proprietary. However, the drawbacks of dynamic analysis with respect to static analysis include checking of one path at a time, consuming lots of system resources and do not have the ability to check the correctness of code. [57]

Before explaining dynamic analysis in terms of UEFI, I would go through a simple example that performs dynamic analysis on windows application. Dr. Memory is a dynamic memory analysis tool for a 32 bit windows application. A 32 bit application can be run with Dr. Memory and the following issues are detected: [58]

1. Unaddressable Access
2. Uninitialized Read
3. Invalid Heap Argument
4. Memory Leaks
5. GDI Usage Errors
6. Handle Leaks
7. Warning

![Figure 5.6: Application with Dr. Memory](image)

After sending the application to Dr. Memory it opens up and the user can traverse through the options that the application allows. The application will respond very slowly and may even crash at times. This is one of the drawbacks of dynamic analysis. In the background Dr. Memory accumulates all the errors generated during the usage of
application. Once the application is closed, a result.txt log file opens, showing the memory leaks/errors/warnings that occurred during the execution of the application.

Dynamic analysis with respect to UEFI was an optional part of my thesis. However, I will try to briefly explain the process that can be followed in order to implement dynamic analysis for UEFI. UEFI runs before the OS is available. The above given example of Dr. Memory is a dynamic analysis tool that runs on the OS. This is a major show stopper in performing dynamic analysis for UEFI. Nevertheless, there seems to be a workaround. An emulator can be used to emulate UEFI on top of an OS. QEMU has the ability to emulate UEFI. These emulators are usually very slow. On top of it if we wish to perform dynamic analysis that could be a real test of patience.

![Diagram of Avatar architecture](image)

**Figure 5.7: Avatar architecture [59]**

*Avatar* is a dynamic firmware analysis tool. It acts as a middle ware between the hardware and emulator allowing the I/O operations to be performed on the hardware and abstracting the firmware by emulating it. [60] QEMU emulator can be used to emulate UEFI. The target device is already the one on which the OS is running. This should be the platform on which the analyzed UEFI code is meant for. *Avatar* should now be configured such that it is stable with the QEMU emulator as well as the platform. Once this has been done, the analysis script should be run in order to perform dynamic analysis on UEFI. The scripts are available in python. The researchers and
developers may write their own scripts to extend the capability of the dynamic firmware analyzer. As mentioned earlier the analysis may be very time consuming and may crash from time to time.

5.3 Summary
During the course of this chapter, we went through the basics of static and dynamic analysis. The concept of these code analysis techniques were discussed as well as their advantages and disadvantages.

3 static analysis tools were discussed, namely; PVS-Studio, Cppcheck and CppDepend. PVS-Studio and CppDepend are proprietary whereas Cppcheck is open source. CppDepend is more of a complexity, architectural and design analysis oriented whereas Cppcheck and PVS-Studio give more code oriented issues. At least Cppcheck should be used by the UEFI developers to improve the code. If the developers are willing to allocate budget to enhance the quality of their code, they can compare and buy a proprietary static analysis tool such as PVS-Studio, Coverity, QA-C and LDRA Testbed. Each static analysis tool should give a unique error and hence the usage of more than one tool is advised. According to John Carmack: “I feel the success that we have had with code analysis has been clear enough that I will say plainly it is irresponsible to not use it.” [55]

Though dynamic analysis of firmware is a tough subject we have given a basic idea on how this could be made possible with respect to UEFI. Avatar is a dynamic firmware analysis tool that sits between an emulator and the hardware to perform dynamic analysis. This tool shall be used to perform dynamic analysis. Crashing and slow speed is a concern while performing dynamic analysis. This concern may amplify for UEFI.
Chapter 6

6. Results
The final chapter will be focused on the results of all 3 phases of my thesis (Available UEFI test tools, test driven development for UEFI and code analysis for UEFI). We will look at the results of each of these phases one by one and discuss how useful they are to bring UEFI to its maturity. Finally, the possibilities of future research will be discussed. My thesis should serve as a starting point for some of these research ideas and as a reference document for the others.

6.1 Results Criteria
The results will be considered separately for the 3 phases; Available UEFI test tools, test driven development for UEFI and code analysis for UEFI. The 3 available tools used to cover the testing of UEFI phases were; Firmware Test Suite, Platform Initialization Self-Certification Test and Chipsec. [61] [62] [63]The unit test which was used for implementing test driven development in UEFI was Unity. The 3 static analysis tools that were discussed are; PVS-Studio, Cppcheck and CppDepend. [64] [65] [66]The tests and analysis were performed on the same project. The dynamic analysis tool that can possibly be used to analyze UEFI is Avatar which requires an emulator such as QEMU. The results criteria will be based on the desired characteristics of the thesis:

1. Increase test coverage
2. Maximum automation of tests
3. Portability of the test suites on other platforms
4. Check against UEFI specification
5. Check protocols that are not part of UEFI specification
6. Capability of integration into regression test framework
6.1.1 Available UEFI Test Tools Results

As discussed in the state of the art, there has been a little focus on UEFI testing historically. Major UEFI tests that are performed in the industry today are manual tests. There are some serious problems in this approach. Manual tests consume a lot of time, are prone to errors, require the tester to have a prerequisite knowledge and have very minimal test coverage.

As part of my thesis 3 test tools were setup. Firmware test suite (fwts) runs on Ubuntu OS and mostly covers the UEFI phases; Transient System Load (TSL), Runtime (RT) and After Life (AL). Platform Initialization Self-Certification Test (PI SCT) runs from the UEFI shell and mostly covers the UEFI phases; Pre EFI Initialization (PEI), Driver Execution Environment (DXE) and partially Boot Device Selection (BDS). Chipsec looks at Security oriented issues. As a result all the 3 tools ensure almost full coverage of UEFI phases. Following table shows a rough estimate of current manual testing versus automated testing:

<table>
<thead>
<tr>
<th></th>
<th>Current Manual Tests</th>
<th>fwts</th>
<th>PI SCT</th>
<th>Chipsec</th>
<th>Combined Automated Test Suites</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Approximate Time</strong></td>
<td>4 weeks</td>
<td>5 minutes</td>
<td>4 hours</td>
<td>1 minute</td>
<td>4 hours and 6 minutes</td>
</tr>
<tr>
<td><strong>Required</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Approximate Number</strong></td>
<td>32</td>
<td>1,182</td>
<td>10,737</td>
<td>9</td>
<td>11,928</td>
</tr>
<tr>
<td><strong>of Test Cases</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Approximate Number</strong></td>
<td>1</td>
<td>74</td>
<td>18</td>
<td>1</td>
<td>93</td>
</tr>
<tr>
<td><strong>of Failed Tests</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Manual testing vs Automated testing for UEFI

This proves that the regression testing time has been decreased from 160 hours to 4 hours i.e. 40 times more efficient. The number of tests has increased from 32 to 11,928 i.e. 373 times more. The number of failed tests has increased from 1 to 93 i.e. the detected issues are 93 times more. The manual tests for UEFI have a very little coverage. They can test very specific tasks such as modifying the options.
from UEFI setup menu and testing if they had an effect. Different OS can be installed to see if they run. UEFI versions can be updated to verify that the updating functionality is successful. Checking whether the ports are initialized and usable. The manual tests and diagnostic tools verify a very minor subset of DXE, BDS, TSL, RT and AL phases. Automated tests certainly increase the test coverage by orders of magnitude because they verify most of the functionalities in all phases of UEFI (SEC, PEI, DXE, BDS, TSL, RT and AL).

The 3 test tools fulfill our results criteria that are based on desired characteristics. They increase the test coverage to full span of UEFI phases. They decrease the testing time because the tools are automated and require little background knowledge of the tester. These test tools should work on a number of platforms. *PI SCT* verifies the implementation according to the UEFI specifications and the source code allows the developers to write their own tests that are not part of the UEFI specification. Moreover, proprietary implementation can also be tested using python scripts and merging them in *Chipsec*. Finally, all these 3 test tools can be integrated into *Software Testing Automation Framework (STAF)* and *STAF eXecution engine (STAX)* by running them from a hard drive. I name STAF and STAX explicitly because another Master thesis at GE IP was conducted to automate tests using STAF and STAX. The Master theses at GE IP end up having a good cohesion and thus proof to be a working industrial solution.

### 6.1.2 Test Driven Development for UEFI Results

Test Driven Development (TDD) has been implemented for UEFI but for a specific IBV i.e. Aptio V, AMI’s EDKII solution. *Unity* (a unit test framework) was integrated and modified to work for UEFI implementation. This was a tough task when talking in terms of UEFI because, there is no output available as early as in the PEI phase. Thus a function was written to generate the output on the debugger for PEI and DXE phase. For some other phases we may use the UEFI shell to output the tests and results of TDD. As of now this should be the only piece of literature that shows implementation of TDD for UEFI (EDKII) is possible.
TDD should help increase the test coverage to a maximum degree because the developer would have already tested the functionality that is implemented. The test cases written while implementing the UEFI functionalities can be used as regression tests later. These test cases will automatically run when the code is compiled. However, TDD does not affect the final production version because the test cases are in a different source file and should be deleted from the customer’s release version. The developer can test the code which is proprietary and not part of the UEFI specification.

### 6.1.3 Code Analysis for UEFI Results

Code analysis was divided into 2 types; static code analysis and dynamic code analysis. 3 static code analysis were used; *PVS-Studio, Cppcheck* and *CppDepend*. Since UEFI has been implemented in C programming language, as opposed to assembly language for BIOS, code analysis tools for C language can be used. However, the research was focused on understanding how effective they are since UEFI does not only have C source code files but also other metadata files. This was a bit of a problem because the static analysis tools pointed out many false alarms as well. Some of the expected variables and libraries were included in module information (.inf) metadata files and not in the C files. Unarguably this could have not been detected by the analyzers which only analyze the C source files.

<table>
<thead>
<tr>
<th>Approximate Number of Issues Reported</th>
<th>PVS-Studio</th>
<th>Cppcheck</th>
<th>CppDepend</th>
</tr>
</thead>
<tbody>
<tr>
<td>12,558</td>
<td>4,013</td>
<td>4,574</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Approximate Number of Files Analyzed</th>
<th>PVS-Studio</th>
<th>Cppcheck</th>
<th>CppDepend</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,125</td>
<td>2,692</td>
<td>1,549</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Approximate Analysis Time</th>
<th>PVS-Studio</th>
<th>Cppcheck</th>
<th>CppDepend</th>
</tr>
</thead>
<tbody>
<tr>
<td>17 minutes</td>
<td>8 minutes</td>
<td>96 hours</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2: Static analysis tools results
PVS-Studio and Cppcheck have a similar purpose and point to code specific issues. They may be compared but CppDepend cannot be compared to them as it has a different purpose. CppDepend gives complexity, architecture and design oriented issues hence takes an enormous amount of time for a large project. Though PVS-Studio gives more issues but there are equally large amount of false negatives. Cppcheck analyzes more files than PVS-Studio because PVS-Studio and CppDepend use a build monitor to recognize which files to analyze. Cppcheck on the other hand analyzes all source code files. However, PVS-Studio and Cppcheck are useful for the UEFI developers when used after every code change. They may point to some specific programming mistakes and can be corrected right away.

Static analysis tools would help minimize bugs in the first place. Secondly they are automatic, user friendly tools. They are meant to be used by the developers to enhance the quality of the code. They are IDE independent and thus, can be used for AMI’s Visual eBIOS.

Dynamic code analysis is a difficult and rather unstable way of improving a complex code such as UEFI. However, a possible workflow has been defined to achieve this task. Avatar (a dynamic firmware analyzer) can be used along with QEMU emulator to perform dynamic analysis of UEFI.

6.2 Further Options
With the progress of my thesis a number of interesting ideas came to my mind. However, these ideas seemed to be time consuming. A research could be made in some of these topics as part of a complete thesis. I will utilize this sub chapter to briefly mention further research ideas.

While I was implementing Test Driven Development I made a quick research into the available unit tests for embedded systems. My judgment call pointed me towards Unity, which seemed to be a good decision at the end. However, I have mentioned a number of unit tests for embedded systems that can be looked into to figure out which of those make sense for UEFI. Out of those which make sense with respect to UEFI can be compared on at least these two criteria; require minimum code modification and provide
maximum useful functionality. Furthermore, Unity itself can be extended to Cmock and Ceedling for developer’s convenience and can be a part of this research.

There are a number of static analysis tools available for C code. Likewise there has been a lot of comparative study on these tools. The results seem to be very confusing and scenario dependent. UEFI is an open source industry standard that will mostly remain the same for almost all PC vendors. A research on static analysis tools with respect to UEFI at this stage under this condition should be of great help to the industry. The result criteria should include; user-friendliness, minimum false alarms, maximum errors.

I was lucky enough to get some quick results for my thesis and decided to keep on extending my research. Dynamic analysis for UEFI was one such voluntary research. I did manage to devise a workflow on how to perform dynamic analysis for UEFI. However, there may be a lot of surprises on the way. A complete thesis can be done to perform dynamic analysis on UEFI code. This may be a tough one. The result criteria may include; feasibility and helpfulness of performing dynamic analysis on UEFI code, optimizing the performance of that analysis.

For working on the above mentioned research possibilities my thesis can be used as a starting point.
Chapter 7

7. Conclusion
UEFI is a relatively new standard that replaces 25 years long BIOS. The initial steps must be
correct to ensure that UEFI too at least lives as long as BIOS did. For this we need to focus on
testing UEFI in parallel to its development. Testing UEFI isn’t easy just like other embedded
systems. Moreover, UEFI has its own challenges to add to the difficulty such as a very large
code base, different contributors, and unavailability of output during the early phases etc. This
makes manual testing nearly impossible and of a limited use. The misconception of assuming
hardware diagnostics to test UEFI should be avoided. These diagnostics, similar to manual
testing will pass almost all test cases. The reason for this lack of quality addition is that the
two techniques do not test the UEFI code. Likewise it should be kept in mind that debugging
is not the same as testing and hence there should be dedicated focus on testing UEFI.

Just like any other problem, the existing solutions should be thoroughly searched for and
avoid reinventing the wheel. If one solution is not sufficient to cover the whole problem,
multiple solutions should be applied. Similarly, UEFI phases could have not been covered by
a single test tool. This is why 3 different test tools had to be setup to ensure a full coverage of
UEFI phases. Firmware Test Suite (fwts) covers the later phases of UEFI (TSL, RT and AL).
PI SCT covers the platform initialization specific phases of UEFI (PEI, DXE and BDS).
Chipsec covers the initial phase of UEFI (SEC). The test tools are automated and hence save
time and effort while ensuring an improvement in the UEFI code quality.

Testing starts late in the Software Development Life Cycle (SDLC) and may be compromised
at times. Secondly quality should be ensured by improving the process continuously rather
than just a quality control at the end. This is where Test Driven Development (TDD) comes
into play. TDD ensures that the developers write the test cases before they code. As a result,
everything that is coded has had some amount of testing already. We have shown that TDD is
possible for the implementation of UEFI. A basic ingredient of TDD is a unit test framework.
A number of unit test frameworks are available for C. The one we have used is Unity for
Aptio V (AMI’s UEFI and EDKII solution).
Another way of ensuring a minimization of bugs is to use code analysis techniques i.e. static code analysis and dynamic code analysis. The evolution of BIOS to UEFI has also shifted the programming language from assembly to C. This means that the UEFI developers can also make use of static code analysis which analyzes the code without executing it. There are a number of proprietary static code analysis tools like PVS-Studio and CppDepend but Cppcheck is open source. The static analysis tools may give lots of false alarms. Nevertheless, they should be used and are very helpful to point out some critical mistakes. Dynamic code analysis on the other hand performs analysis while executing the code. The technique could be very unstable and time consuming. For dynamic analysis on UEFI a workflow has been described and could be looked into with greater detail.
Bibliography


[52] Cppcheck, „Cppcheck 1.66,“ Sourceforge.


Appendix A
Critical failures: 1

oops: Found 2 oopses in kernel log.

High failures: 22

securebootcert: The secure boot variable DB not found.

securebootcert: The secure boot variable KEK not found.

dmicheck: Unmatched Chassis Type: SMBIOS Type 3 reports 0x3 'Desktop' ACPI FACP reports 0x2 'Mobile'

dmicheck: Out of range value 0x00 (range allowed 0x00..0x42) while accessing entry 'Chassis Information (Type 3)' @ 0xabec113b, field 'Base Board Type 0', offset 0x15

syntaxcheck: Assembler error in line 12255

syntaxcheck: Compilation aborted early due to a parser detected syntax error.

syntaxcheck: Assembler error in line 221

syntaxcheck: Assembler error in line 205

syntaxcheck: Assembler error in line 358

syntaxcheck: Assembler error in line 381

syntaxcheck: Assembler error in line 546

power_button: Did not detect any ACPI power buttons events while waiting for power button to be pressed.
method: Detected error 'No return value' when evaluating \_SB\_PCI0.XHC\_RHUB.HS10\_PLD'.

method: Detected error 'No return value' when evaluating \_SB\_PCI0.XHC\_RHUB.HS11\_PLD'.

method: Detected error 'No return value' when evaluating \_SB\_PCI0.XHC\_RHUB.HS12\_PLD'.

method: Detected error 'No return value' when evaluating \_SB\_PCI0.XHC\_RHUB.HS13\_PLD'.

method: Detected error 'No return value' when evaluating \_SB\_PCI0.XHC\_RHUB.HS14\_PLD'.

method: Detected error 'No return value' when evaluating \_SB\_PCI0.XHC\_RHUB.HS15\_PLD'.

brightness: Did not detect ACPI hotkey event.

ac_adapter: Failed to detect an ac_adapter on-line state.

ac_adapter: Did not detect any ACPI ac-adapter events while waiting for power to be disconnected.

ac_adapter: Did not detect any ACPI ac-adapter events while waiting for power to be re-connected.

Medium failures: 27

dmicheck: String index 0x03 in table entry 'System Information (Type 1)' @ 0xabec1043, field 'Version', offset 0x06 has a default value 'To be filled by O.E.M.' and probably has not been updated by the BIOS vendor.
dmicheck: String index 0x04 in table entry 'System Information (Type 1)' @ 0xabc1043, field 'Serial Number', offset 0x07 has a default value 'To be filled by O.E.M.' and probably has not been updated by the BIOS vendor.

dmicheck: String index 0x03 in table entry 'Base Board Information (Type 2)' @ 0xabc10c5, field 'Version', offset 0x06 has a default value 'To be filled by O.E.M.' and probably has not been updated by the BIOS vendor.

dmicheck: String index 0x04 in table entry 'Base Board Information (Type 2)' @ 0xabc10c5, field 'Serial Number', offset 0x07 has a default value 'To be filled by O.E.M.' and probably has not been updated by the BIOS vendor.

dmicheck: String index 0x01 in table entry 'Chassis Information (Type 3)' @ 0xabc113b, field 'Manufacturer', offset 0x04 has a default value 'To Be Filled By O.E.M.' and probably has not been updated by the BIOS vendor.

dmicheck: String index 0x02 in table entry 'Chassis Information (Type 3)' @ 0xabc113b, field 'Version', offset 0x06 has a default value 'To Be Filled By O.E.M.' and probably has not been updated by the BIOS vendor.

dmicheck: String index 0x03 in table entry 'Chassis Information (Type 3)' @ 0xabc113b, field 'Serial Number', offset 0x07 has a default value 'To Be Filled By O.E.M.' and probably has not been updated by the BIOS vendor.

dmicheck: String index 0x04 in table entry 'Chassis Information (Type 3)' @ 0xabc113b, field 'Asset Tag', offset 0x08 has a default value 'To Be Filled By O.E.M.' and probably has not been updated by the BIOS vendor.

microcode: The kernel did not report that CPU 0 has had a microcode update. The current firmware is revision 0x17 and probably has not been updated.

microcode: The kernel did not report that CPU 1 has had a microcode update. The current firmware is revision 0x17 and probably has not been updated.
microcode: The kernel did not report that CPU 2 has had a microcode update. The current firmware is revision 0x17 and probably has not been updated.

microcode: The kernel did not report that CPU 3 has had a microcode update. The current firmware is revision 0x17 and probably has not been updated.

syntaxcheck: Assembler warning in line 61

syntaxcheck: Assembler warning in line 121

syntaxcheck: Assembler warning in line 223

syntaxcheck: Assembler warning in line 294

syntaxcheck: Assembler warning in line 341

syntaxcheck: Assembler warning in line 388

syntaxcheck: Assembler warning in line 435

syntaxcheck: Assembler warning in line 482

syntaxcheck: Assembler warning in line 529

method: Method _SB_.TPM._STR did not return ACPI_TYPE_STRING.

method: _SB_.PCI0.DOCK._EJ0 returned values, but was expected to return nothing.

method: _PR_.CPU0._PCT returned a NULL object, and did not return ACPI_TYPE_PACKAGE.

fan: Fan present but has no cur_state present.

brightness: Backlight acpi_video0 was NOT set to bright level.

brightness: Backlight acpi_video0 was NOT observed going from dim to bright.
Low failures: 22

klog: LOW Kernel message: [ 10.200882] ACPI Warning: 0x00000000000000428-0x0000000000000042f SystemIO conflicts with Region \PMIO 1 (20131115/utaddress-251)

klog: LOW Kernel message: [ 10.200900] ACPI Warning: 0x00000000000000530-0x0000000000000053f SystemIO conflicts with Region \GPRL 1 (20131115/utaddress-251)

klog: LOW Kernel message: [ 10.200904] ACPI Warning: 0x00000000000000530-0x0000000000000053f SystemIO conflicts with Region \GPR_2 (20131115/utaddress-251)

klog: LOW Kernel message: [ 10.200909] ACPI Warning: 0x00000000000000500-0x0000000000000052f SystemIO conflicts with Region \GPRL 1 (20131115/utaddress-251)

klog: LOW Kernel message: [ 10.200913] ACPI Warning: 0x00000000000000500-0x0000000000000052f SystemIO conflicts with Region \GPR_2 (20131115/utaddress-251)

klog: LOW Kernel message: [ 10.200918] lpc_ich: Resource conflict(s) found affecting gpio_ich

dmicheck: String index 0x05 in table entry 'System Information (Type 1)' @ 0xabec1043, field 'SKU Number', offset 0x19 has a default value 'To be filled by O.E.M.' and probably has not been updated by the BIOS vendor.

dmicheck: String index 0x06 in table entry 'Base Board Information (Type 2)' @ 0xabec10c5, field 'Location In Chassis', offset 0x0a has a default value 'To be filled by O.E.M.' and probably has not been updated by the BIOS vendor.

dmicheck: String index 0x05 in table entry 'Chassis Information (Type 3)' @ 0xabec113b, field 'SKU Number', offset 0x18 has a default value 'To be filled by O.E.M.' and probably has not been updated by the BIOS vendor.

dmicheck: String index 0x01 in table entry 'System Configuration Options (Type 12)' @ 0xabec152b, field 'Option 1', offset 0x04 has a default value 'To Be Filled By O.E.M.' and probably has not been updated by the BIOS vendor.
dmicheck: String index 0x04 in table entry 'Processor Information (Type 4)' @ 0xabec15aa, field 'Asset Tag', offset 0x21 has a default value 'Fill By OEM' and probably has not been updated by the BIOS vendor.

dmicheck: String index 0x05 in table entry 'Processor Information (Type 4)' @ 0xabec15aa, field 'Part Number', offset 0x22 has a default value 'Fill By OEM' and probably has not been updated by the BIOS vendor.

dmicheck: String index 0x05 in table entry 'Memory Device (Type 17)' @ 0xabec164e, field 'Asset Tag', offset 0x19 has a default value '9876543210' and probably has not been updated by the BIOS vendor.

dmicheck: String index 0x05 in table entry 'Memory Device (Type 17)' @ 0xabec16f7, field 'Asset Tag', offset 0x19 has a default value '9876543210' and probably has not been updated by the BIOS vendor.

syntaxcheck: Assembler remark in line 131
syntaxcheck: Assembler remark in line 150
syntaxcheck: Assembler remark in line 153
syntaxcheck: Assembler remark in line 183
syntaxcheck: Assembler remark in line 263
syntaxcheck: Assembler remark in line 275

ac_adapter: Failed to detect any state in the ac_adapter state info.

Test | Pass | Fail | Abort | Warn | Skip | Info |
-----------------------------------------------
Total: | 952 | 74 | 5 | 19 | 117 | 15 |
-----------------------------------------------
Appendix B

Failed Tests
Service\Protocol Name
GenericTest\EFICompliantTest
BootServicesTest\MiscBootServicesTest
DevicePathProtocols\DevicePathToTextProtocolTest
DevicePathProtocols\DevicePathFromTextProtocolTest
DevicePathProtocols\DevicePathFromTextProtocolTest
HII\HIIConfigAccessProtocolTest
HII\HIIConfigAccessProtocolTest
HII\HIIConfigAccessProtocolTest
HII\HIIConfigAccessProtocolTest
HII\HIIConfigAccessProtocolTest
HII\HIIConfigAccessProtocolTest
HII\HIIConfigAccessProtocolTest
HII\HIIConfigRoutingProtocolTest
ConsoleSupportTest\SimpleTextInputExProtocolTest

Self Certification Test Report
Service\Protocol Name Total Failed Passed
GenericTest\EFICompliantTest 17 1 16
GenericTest\FloatingPointABITest 2 0 2
BootServicesTest\EventTimerandPriorityServicesTest 33 0 33
BootServicesTest\MemoryAllocationServicesTest 136 0 136
BootServicesTest\ProtocolHandlerServicesTest 1188 0 1188
BootServicesTest\ImageServicesTest 122 0 122
BootServicesTest\MiscBootServicesTest 132 1 131
RuntimeServicesTest\VariableServicesTest 66 0 66
RuntimeServicesTest\TimeServicesTest 84 0 84
RuntimeServicesTest\MiscRuntimeServicesTest 12 0 12
LoadedImageProtocolTest 5020 0 5020
<table>
<thead>
<tr>
<th>Test Case</th>
<th>Pass</th>
<th>Fail</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>DevicePathProtocol\DevicePathProtocolTest</td>
<td>215</td>
<td>0</td>
<td>215</td>
</tr>
<tr>
<td>DevicePathProtocol\DevicePathToTextProtocolTest</td>
<td>21</td>
<td>0</td>
<td>21</td>
</tr>
<tr>
<td>DevicePathProtocol\DevicePathFromTextProtocolTest</td>
<td>43</td>
<td>2</td>
<td>41</td>
</tr>
<tr>
<td>ACPITableProtocolTest</td>
<td>51</td>
<td>2</td>
<td>49</td>
</tr>
<tr>
<td>DriverModelTest\PlatformDriverOverrideProtocolTest</td>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>DriverModelTest\DriverDiagnostics2ProtocolTest</td>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>DriverModelTest\ComponentName2ProtocolTest</td>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>BootableImageSupportTest\SimpleFileSystemProtocolTest</td>
<td>218</td>
<td>0</td>
<td>218</td>
</tr>
<tr>
<td>BootableImageSupportTest\DiskIOProtocolTest</td>
<td>206</td>
<td>0</td>
<td>206</td>
</tr>
<tr>
<td>BootableImageSupportTest\BlockIOProtocolTest</td>
<td>12</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>BootableImageSupportTest\UnicodeCollation2ProtocolTest</td>
<td>36</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>HII\HIDatabaseProtocolTest</td>
<td>50</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>HII\HIIStringProtocolTest</td>
<td>36</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>HII\HIIImageProtocolTest</td>
<td>30</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>HII\HIIFontProtocolTest</td>
<td>37</td>
<td>0</td>
<td>37</td>
</tr>
<tr>
<td>HII\HIIConfigAccessProtocolTest</td>
<td>88</td>
<td>10</td>
<td>78</td>
</tr>
<tr>
<td>PCI\PCI\PCI\PCI\PCIRootBridgeIOProtocolTest</td>
<td>30</td>
<td>1</td>
<td>29</td>
</tr>
<tr>
<td>PCI\PCI\PCI\PCI\PCI\IO\PCI\IO\ProtocolTest</td>
<td>58</td>
<td>0</td>
<td>58</td>
</tr>
<tr>
<td>USB\USB\USB\USB\USBIOProtocolTest</td>
<td>2235</td>
<td>0</td>
<td>2235</td>
</tr>
<tr>
<td>DebuggerSupportTest\DebugSupportProtocolTest</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>CompressionTest\DecompressProtocolTest</td>
<td>12</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>EFI\ByteCodeTest\EBC\InterpreterProtocolTest</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>ConsoleSupportTest\SimpleTextInputExProtocolTest</td>
<td>44</td>
<td>1</td>
<td>43</td>
</tr>
<tr>
<td>ConsoleSupportTest\SimpleInputProtocolTest</td>
<td>16</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>ConsoleSupportTest\SimpleOutputProtocolTest</td>
<td>64</td>
<td>0</td>
<td>64</td>
</tr>
<tr>
<td>ConsoleSupportTest\SimplePointerProtocolTest</td>
<td>18</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>ConsoleSupportTest\GraphicsOutputProtocolTest</td>
<td>20</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>Total</td>
<td>10737</td>
<td>18</td>
<td>10719</td>
</tr>
</tbody>
</table>
Appendix C

[+] imported platform specific configuration: chipsec.cfg.hsw

OS      : uefi
Platform: Desktop 4th Generation Core Processor (Haswell CPU / Lynx Point PCH)
  VID: 8086
  DID: 0C00
CHIPSEC : 1.1.0

[*] loading common modules from "\chipsec\modules\common" ..
[+] loaded chipsec.modules.common.bios_kbrd_buffer
[+] loaded chipsec.modules.common.bios_ts
[+] loaded chipsec.modules.common.bios_wp
[+] loaded chipsec.modules.common.smm
[+] loaded chipsec.modules.common.smrr
[+] loaded chipsec.modules.common.spi_lock
[+] loaded chipsec.modules.common.secureboot.keys
[+] loaded chipsec.modules.common.secureboot.variables
[*] loading platform specific modules from ".\chipsec\modules\hsw" ..
[+] loading modules from ".\chipsec\modules" ..
[+] loaded chipsec.modules.module_template
[*] running loaded modules ..

[+] imported chipsec.modules.common.bios_kbrd_buffer
[*] Module path: chipsec/modules/common/bios_kbrd_buffer.pyc
[x][=======================================================================
[x][ Test: Pre-boot Passwords in the BIOS Keyboard Buffer
[x][=======================================================================
[*] Keyboard buffer head pointer = 0x1E (at 0x41A), tail pointer = 0x1E (at 0x41C)
[*] Keyboard buffer contents (at 0x41E):
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
[*] Checking contents of the keyboard buffer..

[+] PASSED: Keyboard buffer looks empty. Pre-boot passwords don't seem to be exposed

[+] imported chipsec.modules.common.bios_ts
!*] Module path: chipsec/modules/common/bios_ts.pyc
[x][=======================================================================
[x][ Test: BIOS Interface Lock and Top Swap Mode
[x][=======================================================================
[*] RCBA General Config base: 0xFED1F400
[*] GCS (General Control and Status) register = 0x00000C01
[10] BBS (BIOS Boot Straps) = 0x3
[00] BILD (BIOS Interface Lock-Down) = 1
[*] BUC (Backed Up Control) register = 0x00000000
[00] TS (Top Swap) = 0
[*] BC (BIOS Control) register = 0x00
[04] TSS (Top Swap Status) = 0
[*] BIOS Top Swap mode is disabled

[+] PASSED: BIOS Interface is locked (including Top Swap Mode)

[+] imported chipsec.modules.common.bios_wp
[*] Module path: chipsec/modules/common/bios_wp.pyc
[x][ ] Test: BIOS Region Write Protection
[x][ ] BIOS Control (BDF 0:31:0 + 0xDC) = 0x00
[05] SMM_BWP = 0 (SMM BIOS Write Protection)
[04] TSS = 0 (Top Swap Status)
[01] BLE = 0 (BIOS Lock Enable)
[00] BIOSWE = 0 (BIOS Write Enable)

[-] BIOS region write protection is disabled!

[*] BIOS Region: Base = 0x00A00000, Limit = 0x00FFFFFF
SPI Protected Ranges

<table>
<thead>
<tr>
<th>PRx (offset)</th>
<th>Value</th>
<th>Base</th>
<th>Limit</th>
<th>WP?</th>
<th>RP?</th>
</tr>
</thead>
<tbody>
<tr>
<td>PR0 (74)</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PR1 (78)</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PR2 (7C)</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PR3 (80)</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PR4 (84)</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

[!] None of the SPI protected ranges write-protect BIOS region

[!] BIOS should enable all available SMM based write protection mechanisms or configure SPI protected ranges to protect the entire BIOS region
[-] FAILED: BIOS is NOT protected completely

[+] imported chipsec.modules.common.smm
[*] Module path: chipsec/modules/common/smm.pyc
[x][ ] Test: Compatible SMM memory (SMRAM) Protection
[x][ ] Compatible SMRAM Control (00:00.0 + 0x88) = 0x1A
[06] D_OPEN = 0 (SMRAM Open)
[05] D_CLS = 0 (SMRAM Closed)
[04] D_LCK = 1 (SMRAM Locked)
[03] G_SMRAME = 1 (SMRAM Enabled)
[02:00] C_BASE_SEG = 2 (SMRAM Base Segment = 010b)

[*] Compatible SMRAM is enabled  
[+] PASSED: Compatible SMRAM is locked down

[+] imported chipsec.modules.common.smrr
[*] Module path: chipsec/modules/common/smrr.pyc
[x]========================================================================
[x] Test: CPU SMM Cache Poisoning / SMM Range Registers (SMRR)
[x]========================================================================
[+] OK. SMRR are supported in IA32_MTRRCAP_MSR

[*] Checking SMRR Base programming..  
[*] IA32_SMRR_BASE_MSR = 0x00000000AC000006  
BASE = 0xAC000000  
MEMTYPE = 6  
[+] SMRR Memtype is WB
[+] OK so far. SMRR Base is programmed

[*] Checking SMRR Mask programming..  
[*] IA32_SMRR_MASK_MSR = 0x00000000FF000800  
MASK = 0xFF000000  
VLD = 1  
[+] OK so far. SMRR are enabled in SMRR_MASK MSR

[*] Verifying that SMRR_BASE/MASK have the same values on all logical CPUs..  
ERROR: Exception occurred during chipsec.modules.common.smrr.run(): 'EfiHelper instance has no attribute 'get_threads_count''

[+] imported chipsec.modules.common.spi_lock
[*] Module path: chipsec/modules/common/spi_lock.pyc
[x]========================================================================
[x] Test: SPI Flash Controller Configuration Lock
[x]========================================================================
[*] HSFS register = 0x0000F008  
FLOCKDN = 1  
[+] PASSED: SPI Flash Controller configuration is locked

[+] imported chipsec.modules.common.secureboot.keys
[*] Module path: chipsec/modules/common/secureboot/keys.pyc
[x]========================================================================
[x] Test: Protection of Secure Boot Key and Configuration EFI Variables
 Chipsec results log

[*] SKIPPED: Currently this module can only run on Windows 8 or greater or Linux. Exiting..
[+] imported chipsec.modules.common.secureboot.variables
[*] Module path: chipsec/modules/common/secureboot/variables.pyc
[x][ =======================================================================
[x][*] Test: Attributes of Secure Boot EFI Variables
[x][ =======================================================================
[*] SKIPPED: Currently this module can only run on Windows 8 or higher or Linux. Exiting..
[+] imported chipsec.modules.module_template
[*] Module path: chipsec/modules/module_template.pyc
[x][ =======================================================================
[x][ Test: Module Template
[x][ =======================================================================
[+] PASSED: Test Passed

[CHIPSEC] ************************************** SUMMARY *****************************
[CHIPSEC] Time elapsed          0.000
[CHIPSEC] Modules total         9
[CHIPSEC] Modules failed to run 1:
ERROR: chipsec.modules.common.smrr
[CHIPSEC] Modules passed        5:
[+] PASSED: chipsec.modules.common.bios_kbrd_buffer
[+] PASSED: chipsec.modules.common.bios_ts
[+] PASSED: chipsec.modules.common.smm
[+] PASSED: chipsec.modules.common.spi_lock
[+] PASSED: chipsec.modules.module_template
[CHIPSEC] Modules failed        1:
[-] FAILED: chipsec.modules.common.bios_wp
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 2:
[*] SKIPPED: chipsec.modules.common.secureboot.keys
[*] SKIPPED: chipsec.modules.common.secureboot.variables
[CHIPSEC] ************************************************************
[CHIPSEC] Version:   1.1.0
Appendix D

GE\GE_Features\GE_EcPkg\GE_EcDxe\GE_EcDxe.c:751]: (style) The scope of the variable 'FilePath' can be reduced.

GE\GE_Features\GE_EcPkg\GE_EcDxe\GE_EcDxe.c:505]: (style) Variable 'Status' is assigned a value that is never used.

GE\GE_Features\GE_EcPkg\GE_EcDxe\GE_EcDxe.c:441]: (style) Variable 'Data32' is assigned a value that is never used.

GE\GE_Features\GE_EcPkg\GE_EcDxe\GE_EcDxe.c:707]: (performance) Variable 'Status' is reassigned a value before the old one has been used.

GE\GE_Features\GE_EcPkg\GE_EcDxe\GE_EcDxe.c:711]: (performance) Variable 'Status' is reassigned a value before the old one has been used.

GE\GE_Features\GE_EcPkg\GE_EcDxe\GE_EcDxe.c:835]: (performance) Variable 'Status' is reassigned a value before the old one has been used.

GE\GE_Features\GE_EcPkg\GE_EcDxe\GE_EcDxe.c:848]: (performance) Variable 'Status' is reassigned a value before the old one has been used.

GE\GE_Features\GE_EcPkg\GE_EcDxe\GE_EcProtocol.c:156]: (performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

GE\GE_Features\GE_EcPkg\GE_EcDxe\GE_EcProtocol.c:158]: (performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

GE\GE_Features\GE_EcPkg\GE_EcDxe\GE_EcProtocol.c:159]: (performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

GE\GE_Features\GE_EcPkg\GE_EcDxe\GE_EcProtocol.c:330]: (performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

GE\GE_Features\GE_EcPkg\GE_EcDxe\GE_EcProtocol.c:332]: (performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

GE\GE_Features\GE_EcPkg\GE_EcDxe\GE_EcProtocol.c:378]: (performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

GE\GE_Features\GE_EcPkg\GE_EcDxe\GE_EcProtocol.c:404]: (performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

GE\GE_Features\GE_EcPkg\GE_EcFdOvrStrap\GE_EcFdOvrStrapDxe\GE_EcFdOvrStrapDxeEntry.c:83]: (performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

GE\GE_Features\GE_EcPkg\GE_EcFdOvrStrap\GE_EcFdOvrStrapDxe\GE_EcFdOvrStrapDxeEntry.c:92]: (performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.
[GE\GE_Features\GE_EcPkg\GE_EcMdOvrStrap\GE_EcMdOvrStrapDxe\GE_EcMdOvrStrapP rotocol.c:43] ->
[GE\GE_Features\GE_EcPkg\GE_EcMdOvrStrap\GE_EcMdOvrStrapDxe\GE_EcMdOvrStrapP rotocol.c:49]: (performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_EcPkg\GE_EcMdOvrStrap\GE_EcMdOvrStrapDxe\GE_EcMdOvrStrapP rotocol.c:69] ->
[GE\GE_Features\GE_EcPkg\GE_EcMdOvrStrap\GE_EcMdOvrStrapDxe\GE_EcMdOvrStrapP rotocol.c:75]: (performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_EcPkg\GE_EcMdOvrStrap\GE_EcMdOvrStrapDxe\GE_EcMdOvrStrapP rotocol.c:94] ->
[GE\GE_Features\GE_EcPkg\GE_EcMdOvrStrap\GE_EcMdOvrStrapDxe\GE_EcMdOvrStrapP rotocol.c:100]: (performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_EcPkg\GE_EcMdOvrStrap\GE_EcMdOvrStrapDxe\GE_EcMdOvrStrapLib\GE_EcMdOvrStrapLib.c:137]: (style) Variable 'Buffer8' is assigned a value that is never used.

[GE\GE_Features\GE_EcPkg\GE_EcMdOvrStrap\GE_EcMdOvrStrapPei\GE_EcMdOvrStrapPeiEntry.c:94] ->
[GE\GE_Features\GE_EcPkg\GE_EcMdOvrStrap\GE_EcMdOvrStrapPei\GE_EcMdOvrStrapPeiEntry.c:102]: (performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_EcPkg\GE_EcMdOvrStrap\GE_EcMdOvrStrapPei\GE_EcMdOvrStrapP ei.c:74] ->
[GE\GE_Features\GE_EcPkg\GE_EcMdOvrStrap\GE_EcMdOvrStrapPei\GE_EcMdOvrStrapP ei.c:80]: (performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_EcPkg\GE_EcMdOvrStrap\GE_EcMdOvrStrapPei\GE_EcMdOvrStrapP ei.c:100] ->
[GE\GE_Features\GE_EcPkg\GE_EcMdOvrStrap\GE_EcMdOvrStrapPei\GE_EcMdOvrStrapP ei.c:106]: (performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_EcPkg\GE_EcMdOvrStrap\GE_EcMdOvrStrapPei\GE_EcMdOvrStrapP ei.c:125] ->
[GE\GE_Features\GE_EcPkg\GE_EcMdOvrStrap\GE_EcMdOvrStrapPei\GE_EcMdOvrStrapP ei.c:131]: (performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.
[GE\GE_Features\GE_EcPkg\GE_EcKbc\GE_EcKbcDxe\GE_EcKbcDxe.c:175] ->
[GE\GE_Features\GE_EcPkg\GE_EcKbc\GE_EcKbcDxe\GE_EcKbcDxe.c:201]:
(performance) Variable 'Status' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_EcPkg\GE_EcKbc\GE_EcKbcPei\GE_EcKbcPei.c:192]: (style)
Variable 'Status' is assigned a value that is never used.

[GE\GE_Features\GE_EcPkg\GE_EcOofbdFwUpdate\GE_EcOofbdFwUpdate.c:389]: (style)
The scope of the variable 'Data32' can be reduced.

[GE\GE_Features\GE_EcPkg\GE_EcOofbdFwUpdate\GE_EcOofbdFwUpdate.c:526]: (style)
The scope of the variable 'pOFBDHdr' can be reduced.

[GE\GE_Features\GE_EcPkg\GE_EcOofbdFwUpdate\GE_EcOofbdFwUpdate.c:527]: (style)
The scope of the variable 'pOFBDExtHdr' can be reduced.

[GE\GE_Features\GE_EcPkg\GE_EcOofbdFwUpdate\GE_EcOofbdFwUpdate.c:528]: (style)
The scope of the variable 'pOFBDTblEnd' can be reduced.

[GE\GE_Features\GE_EcPkg\GE_EcOofbdFwUpdate\GE_EcOofbdFwUpdate.c:529]: (style)
The scope of the variable 'ECStructPtr' can be reduced.

[GE\GE_Features\GE_EcPkg\GE_EcOofbdFwUpdate\GE_EcOofbdFwUpdate.c:536]: (style)
Variable 'pOFBDTblEnd' is assigned a value that is never used.

[GE\GE_Features\GE_EcPkg\GE_EcPei\GE_EcPei.c:50] ->
[GE\GE_Features\GE_EcPkg\GE_EcPei\GE_EcPei.c:103]: (performance) Variable 'EfiStatus'
is reassigned a value before the old one has been used.

[GE\GE_Features\GE_EcPkg\GE_EcPeiSetMeRunmode\GE_EcPeiSetMeRunmodeEntry.c:47] ->
[GE\GE_Features\GE_EcPkg\GE_EcPeiSetMeRunmode\GE_EcPeiSetMeRunmodeEntry.c:68]: (performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_EcPkg\GE_EcSetup\GE_EcSetup.c:49] ->
[GE\GE_Features\GE_EcPkg\GE_EcSetup\GE_EcSetup.c:65]: (performance) Variable 'Status' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_SmBiosPkg\GE_SmBiosDynUpdt\GE_SmBiosDynUpdt.c:140] ->
[GE\GE_Features\GE_SmBiosPkg\GE_SmBiosDynUpdt\GE_SmBiosDynUpdt.c:148]:
(performance) Variable 'Status' is reassigned a value before the old one has been used.
[GE\GE_Features\GE_SmBiosPkg\GE_SmBiosDynUpdt\GE_SmBiosDynUpdt.c:148] ->
[GE\GE_Features\GE_SmBiosPkg\GE_SmBiosDynUpdt\GE_SmBiosDynUpdt.c:149]:
(performance) Variable 'Status' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_SmBiosPkg\GE_SmBiosDynUpdt\GE_SmBiosDynUpdt.c:149] ->
[GE\GE_Features\GE_SmBiosPkg\GE_SmBiosDynUpdt\GE_SmBiosDynUpdt.c:150]:
(performance) Variable 'Status' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_SmBiosPkg\GE_SmBiosDynUpdt\GE_SmBiosDynUpdt.c:150] ->
[GE\GE_Features\GE_SmBiosPkg\GE_SmBiosDynUpdt\GE_SmBiosDynUpdt.c:151]:
(performance) Variable 'Status' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_SmBiosPkg\GE_SmBiosDynUpdt\GE_SmBiosDynUpdt.c:151]:
(style) Variable 'Status' is assigned a value that is never used.

[GE\GE_Features\GE_SmBiosPkg\GE_SmBiosDynUpdt\GE_SmBiosTypeHandlers.c:92] ->
[GE\GE_Features\GE_SmBiosPkg\GE_SmBiosDynUpdt\GE_SmBiosTypeHandlers.c:104]:
(performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_SmBiosPkg\GE_SmBiosDynUpdt\GE_SmBiosTypeHandlers.c:203] ->
[GE\GE_Features\GE_SmBiosPkg\GE_SmBiosDynUpdt\GE_SmBiosTypeHandlers.c:214]:
(performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_SmBiosPkg\GE_SmBiosDynUpdt\GE_SmBiosTypeHandlers.c:332] ->
[GE\GE_Features\GE_SmBiosPkg\GE_SmBiosDynUpdt\GE_SmBiosTypeHandlers.c:343]:
(performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_SmBiosPkg\GE_SmBiosDynUpdt\GE_SmBiosTypeHandlers.c:415] ->
[GE\GE_Features\GE_SmBiosPkg\GE_SmBiosDynUpdt\GE_SmBiosTypeHandlers.c:426]:
(performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtDxe\GE_TcoWdtDxeEntry.c:86] ->
[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtDxe\GE_TcoWdtDxeEntry.c:145]:
(performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtDxe\GE_TcoWdtDxeEntry.c:156] ->
[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtDxe\GE_TcoWdtDxeEntry.c:170]:
(performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtLib\GE_TcoWdtLib.c:188] ->
[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtLib\GE_TcoWdtLib.c:194]: (performance)
Variable 'EfiStatus' is reassigned a value before the old one has been used.
[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtLib\GE_TcoWdtLib.c:231] ->
[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtLib\GE_TcoWdtLib.c:237]: (performance)
Variable 'EfiStatus' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtLib\GE_TcoWdtLib.c:73]: (style)
Variable 'EfiStatus' is assigned a value that is never used.

[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtPei\GE_TcoWdtPeiEntry.c:50] ->
[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtPei\GE_TcoWdtPeiEntry.c:72]:
(performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtPei\GE_TcoWdtPeiEntry.c:61]: (style)
Variable 'Reg16Val' is assigned a value that is never used.

[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtSetup\GE_TcoWdtSetupHooks.c:368] ->
[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtSetup\GE_TcoWdtSetupHooks.c:373]:
(performance) Variable 'Status' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtSetup\GE_TcoWdtSetupHooks.c:128]:
(style) Variable 'EfiStatus' is assigned a value that is never used.

[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtSetup\GE_TcoWdtSetupHooks.c:175]:
(style) Variable 'EfiStatus' is assigned a value that is never used.

[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtSetup\GE_TcoWdtSetupHooks.c:315]:
(style) Variable 'Status' is assigned a value that is never used.

[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtSetup\GE_TcoWdtSetupHooks.c:380]:
(style) Variable 'Status' is assigned a value that is never used.

[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtSetup\GE_TcoWdtSetupHooksHelpers.c:64] ->
[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtSetup\GE_TcoWdtSetupHooksHelpers.c:74]:
(performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtSetup\GE_TcoWdtSetupHooksHelpers.c:106 ] ->
[GE\GE_Features\GE_TcoWdtPkg\GE_TcoWdtSetup\GE_TcoWdtSetupHooksHelpers.c:116 ]:
(performance) Variable 'EfiStatus' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_TdDxePkg\GE_TddDxeLib\unity.c:140]: (style)
The scope of the variable 'next_divisor' can be reduced.

[GE\GE_Features\GE_TdDxePkg\GE_TddDxeLib\unity.c:173]: (style)
The scope of the variable 'next_divisor' can be reduced.
[GE\GE_Features\GE_TddDxePkg\GE_TddDxeLib\unity.c:197]: (style) The scope of the variable 'nibble' can be reduced.

[GE\GE_Features\GE_TddDxePkg\GE_TddDxeLib\unity.c:548]: (style) The scope of the variable 'diff' can be reduced.

[GE\GE_Features\GE_TddDxePkg\GE_TddDxeLib\unity.c:548]: (style) The scope of the variable 'tol' can be reduced.

[GE\GE_Features\GE_TddDxePkg\GE_TddDxeLib\unity.c:922]: (style) The scope of the variable 'i' can be reduced.

[GE\GE_Features\GE_TddDxePkg\GE_TddDxeLib\unity.c:1026]: (style) The scope of the variable 'bytes' can be reduced.

[GE\GE_Features\GE_TddDxePkg\GE_TddDxeLib\unity.c:720]: (style) The scope of the variable 'tol' can be reduced.

[GE\GE_Features\GE_TddDxePkg\GE_TddDxeLib\unity.c:720]: (style) The scope of the variable 'tol' can be reduced.

[GE\GE_Features\GE_TddPeiPkg\GE_TddPeiLib\unity.c:140]: (style) The scope of the variable 'next_divisor' can be reduced.

[GE\GE_Features\GE_TddPeiPkg\GE_TddPeiLib\unity.c:173]: (style) The scope of the variable 'next_divisor' can be reduced.

[GE\GE_Features\GE_TddPeiPkg\GE_TddPeiLib\unity.c:197]: (style) The scope of the variable 'nibble' can be reduced.

[GE\GE_Features\GE_TddPeiPkg\GE_TddPeiLib\unity.c:548]: (style) The scope of the variable 'diff' can be reduced.

[GE\GE_Features\GE_TddPeiPkg\GE_TddPeiLib\unity.c:548]: (style) The scope of the variable 'tol' can be reduced.

[GE\GE_Features\GE_TddPeiPkg\GE_TddPeiLib\unity.c:922]: (style) The scope of the variable 'i' can be reduced.

[GE\GE_Features\GE_TddPeiPkg\GE_TddPeiLib\unity.c:1026]: (style) The scope of the variable 'bytes' can be reduced.

[GE\GE_Features\GE_TddPeiPkg\GE_TddPeiLib\unity.c:720]: (style) The scope of the variable 'diff' can be reduced.

[GE\GE_Features\GE_TddPeiPkg\GE_TddPeiLib\unity.c:720]: (style) The scope of the variable 'tol' can be reduced.
[GE\GE_Features\GE_TddUefiAppPkg\GE_TddUefiAppLib\unity.c:143]: (style) The scope of the variable 'next_divisor' can be reduced.

[GE\GE_Features\GE_TddUefiAppPkg\GE_TddUefiAppLib\unity.c:176]: (style) The scope of the variable 'next_divisor' can be reduced.

[GE\GE_Features\GE_TddUefiAppPkg\GE_TddUefiAppLib\unity.c:200]: (style) The scope of the variable 'nibble' can be reduced.

[GE\GE_Features\GE_TddUefiAppPkg\GE_TddUefiAppLib\unity.c:551]: (style) The scope of the variable 'diff' can be reduced.

[GE\GE_Features\GE_TddUefiAppPkg\GE_TddUefiAppLib\unity.c:551]: (style) The scope of the variable 'tol' can be reduced.

[GE\GE_Features\GE_TddUefiAppPkg\GE_TddUefiAppLib\unity.c:925]: (style) The scope of the variable 'i' can be reduced.

[GE\GE_Features\GE_TddUefiAppPkg\GE_TddUefiAppLib\unity.c:1029]: (style) The scope of the variable 'bytes' can be reduced.

[GE\GE_Features\GE_TddUefiAppPkg\GE_TddUefiAppLib\unity.c:723]: (style) The scope of the variable 'diff' can be reduced.

[GE\GE_Features\GE_TddUefiAppPkg\GE_TddUefiAppLib\unity.c:723]: (style) The scope of the variable 'tol' can be reduced.

[GE\GE_Features\GE_TseMainInfoPkg\GE_TseMainInfoSetup\GE_TseMainInfoSetupHook s.c:263] ->
[GE\GE_Features\GE_TseMainInfoPkg\GE_TseMainInfoSetup\GE_TseMainInfoSetupHook s.c:275]: (performance) Variable 'Status' is reassigned a value before the old one has been used.

[GE\GE_Features\GE_TseMainInfoPkg\GE_TseMainInfoSetup\GE_TseMainInfoSetupHook s.c:373] ->
[GE\GE_Features\GE_TseMainInfoPkg\GE_TseMainInfoSetup\GE_TseMainInfoSetupHook s.c:375]: (performance) Variable 'Status' is reassigned a value before the old one has been used.