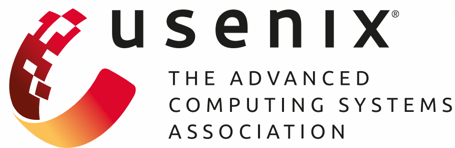

DECAF: Automatic, Adaptive De-bloating and Hardening of COTS Firmware

Jake Christensen, *Private Machines;* Ionut Mugurel Anghel, *Univ. Politehnica Bucharest;* Rob Taglang, *Private Machines;* Mihai Chiroiu, *Univ. Politehnica Bucharest;* Radu Sion, *Private Machines*

https://www.usenix.org/conference/usenixsecurity20/presentation/christensen

This paper is included in the Proceedings of the
29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the
29th USENIX Security Symposium
is sponsored by USENIX.

DECAF: Automatic, Adaptive De-bloating and Hardening of COTS Firmware

Jake Christensen
Private Machines

Ionut Mugurel Anghel
Univ. Politehnica Bucharest

Rob Taglang
Private Machines

Mihai Chiroiu
Univ. Politehnica Bucharest

Radu Sion
Private Machines

Abstract

Once compromised, server firmware can surreptitiously and permanently take over a machine and any stack running thereon, with no hope for recovery, short of hardware-level intervention. To make things worse, modern firmware contains millions of lines of unnecessary code and hundreds of unnecessary modules as a result of a long firmware supply chain designed to optimize time-to-market and cost, but not security. As a result, off-the-shelf motherboards contain large, unnecessarily complex, closed-source vulnerability surfaces that can completely and irreversibly compromise systems.

In this work, we address this problem by dramatically and automatically reducing the vulnerability surface. DECAF is an extensible platform for automatically pruning a wide class of commercial UEFI firmware. DECAF intelligently runs dynamic iterative surgery on UEFI firmware to remove a maximal amount of code with no regressive effects on the functionality and performance of higher layers in the stack (OS, applications).

DECAF has successfully pruned over 70% of unnecessary, redundant, reachable firmware in leading server-grade motherboards with no effect on the upper layers, and increased resulting system performance and boot times.

1 Introduction

Millions of lines of C, assembly, and microcode compose the binaries residing on today's motherboards.

Firmware is essential in managing and running the underlying hardware. Yet, due to the complicated and inherently market-driven process of hardware manufacture and sale, much of the firmware delivered with modern motherboards is not necessary for the hardware on which it ships.

Manually customizing firmware for a given motherboard and application is simply not practical. It can take thousands of hours of work to do right and is not scalable to constantly changing hardware, purchasing decisions, environments, and applications of modern consumers and corporations.

As a result, a typical supply chain for Unified Extensible Firmware Interface (UEFI) firmware starts with EDK II [3], the open source reference UEFI implementation from TianoCore. The EDK II project measures up to roughly 2.5 million lines of code. Vendor specific implementations tend to be even larger. A motherboard firmware company (American Megatrends, Phoenix Technologies, etc.) adds the necessary modules from Intel for a particular chipset along with any other modules needed for their base design. Motherboard manufacturers (Dell, ASUS, etc.) then add further modules required to enable proprietary hardware or management features, further bloating the firmware which ultimately ships with the hardware. More details about the firmware layout and the role of modules are given in Section 2.2

Due to the nature of this supply chain, the firmware trades hands numerous times before it is delivered to a board and ultimately to an end user. At each stage, modules are added to the firmware, but typically, for time and cost reasons, nothing is optimized or removed, including any generic modules that do not apply to the specific hardware being delivered.

Furthermore, firmware fixes are often neglected even for motherboards only 6-12 months old. Worse still, even when acting in good-faith, it is difficult for manufacturers to fix bugs which may originate in a module from an upstream, generic firmware vendor that propagate down to specific motherboards. Addressing this problem is not trivial and places security-conscious users in a difficult position.

Most importantly, very large portions of existing firmware are unnecessary, significantly increasing the vulnerability surface of a system and degrading performance. A bloated firmware code base is not only a problem in terms of performance and boot time, but also has major security implications. A recent study has shown that because of the predictable supply chain, the numerous additional modules in UEFI images, and large amount of code reuse between images, certain attacks can be easily and reliably automated [45].

This is not a problem unique to firmware. In today's highly over-provisioned systems, it is simply cheaper and easier to pile onto an existing code base than to design from the ground

up. Modern software is bloated and routinely uses only a few percentage points of the binary code. A recent study has shown that only 10% of the shared libraries in Ubuntu 16.04 are used by actual programs [32].

To make matters worse, in the case of firmware, exploits can completely compromise an entire system, including any trust chains and security mechanisms such as "secure boot" [11]. Short of physical intervention and hardware reflashing, users are often left with completely insecure systems, without any ability to even detect the breach.

One of the first steps that can be taken is to reduce this vulnerability surface by eliminating any unnecessary bloat. This results in a linear reduction of the overall vulnerability surface and availability of exploits.

In this work we propose to automatically and dynamically prune significant amounts of unnecessary binary code from a large class of COTS firmware without impacting the functionality of the upper layer of the stack (OS, applications).

1.1 UEFI Has a Quality Problem

Bloat is not the only problem with UEFI. There are a great many vulnerabilities in the wild that are completely avoidable, but exist due to manufacturer negligence. Many common attack vectors on UEFI have modern mitigations that manufacturers fail to properly configure.

In a survey of firmware vulnerabilities [29] covering 2015-2017, not only are the total numbers concerning, but there is also an increasing trend in the number of vulnerabilities due to lack of proper configurations of increasingly numerous security options.

Firmware expert Nikolaj Schlej, perhaps best known as the author of the widely used and popular UEFITool [36], has been sounding the alarm for years through various of talks and presentations. For example, in [38] numerous vulnerabilities for off the shelf firmware are introduced. Compelling arguments are made for users to immediately patch their own systems rather than wait for manufacturer firmware updates which may never come and rarely address bugs in time. "[I]f the firmware can still boot your OS - it's fine to have [...] components removed".

Unfortunately, this is easier said than done. For users (either consumer or enterprise) of off-the-shelf firmware, it is effectively a proprietary black box. Users do not have the expertise and tools to properly prune a BIOS. They are thus often left with 3-5 year old firmware with no recourse. **This is one of the main motivators behind DECAF**, namely empowering non-expert users to easily remove old, unwanted or buggy functionality from their firmware.

Since much of the firmware is closed-source, it is difficult to precisely evaluate firmware code quality and whether it is that much better than the abysmal industry average featuring multiple bugs for every hundred lines of code [25].

Yet, analysing open-source Intel code provides some insight into what might be going on behind the scenes [37]. For example, for the Intel Galileo board, using only a static code analyzer restricted to search only for "obviously incorrect code fragments" numerous bugs can be found, which appear to be the result of lazy copy-pasting.

1.2 DECAF

Debloating is perfectly suited to firmware hardening because of the previously described supplier model. If done properly, as a result of the UEFI structure, it can be applied at module granularity to any motherboard, even without access to the source code.

DECAF is an extensible platform for automatically pruning a wide class of commercial UEFI firmware. It utilizes a configurable set of validation tests to tailor the retained functionality to a particular use-case and intelligently performs a dynamic iterative surgery process on UEFI binary firmware to remove a maximal amount of code with no effect on functionality and performance of higher layers in the stack (OS, applications).

DECAF also supports module white and black listing to take advantage of prior knowledge of the target firmware. For example, an in-BIOS DHCP implementation is needed (for example, for PXE boot), and the given firmware contains two implementations: one from the EDK II standard and one from the manufacturer. In this case, we can, for example, black list the implementation from the manufacturer and white list the open source one.

We evaluated DECAF experimentally in two configurations: one targeted at running cloud hypervisors, and one targeted at maximal byte removal (booting off of local media). Results show that up 30% of the codebase can be pruned *automatically* in the first case and up to 70% in the latter with no impact on the upper layers. The resulting firmware boots significantly faster as well.

At first, it may seem that code that does not affect functionality is unreachable, and thus its removal may be of little security benefit. This, however, is not the case. Most firmware contains active, reachable code that is simply unused by the upper layers but poses significant vulnerability challenges (e.g. multiple network stacks, obsolete drivers for tens of peripherals/USB/VGA, entire GUIs, etc.). Indeed, the fact that pruned firmware boots significantly faster than original images is incontrovertible evidence that the execution path is modified. In summary:

1. DECAF is the first extensible platform for automatically pruning commercial UEFI firmware.
2. DECAF can automatically prune up to 70% of a UEFI image.
3. DECAF includes a framework for automatic testing of UEFI images on real boards.

4. DECAF operates on binaries (no need for source code) and can easily integrate with and operate on new motherboards.
5. DECAF has been successfully applied on multiple (6) motherboard lines; more are added periodically.
6. **DECAFed firmware has been successfully running in a production-grade data center environment since mid 2017**.
7. UEFI firmware can be easily customized to retain or remove only desired functionality.

2 Background

2.1 UEFI

UEFI (originally EFI) was developed to replace legacy BIOS with a more standardized solution in order to improve interoperability between vendors.

UEFI splits the lifetime of platform initialization into 4 distinct phases: (1) Security (SEC), (2) Pre-EFI Initialization Environment (PEI), (3) Driver Execution Environment (DXE), (4) Boot Device Selection (BDS).

The SEC stage is the root of trust of the system and does very early hardware initialization and validation of the firmware image. It then bootstraps and hands execution off to the PEI stage. The PEI stage finalizes hardware initialization. It enumerates platform information into a series of Hand Off Blocks (HOBs) that are handed off to the DXE stage. The PEI stage execution is heavily dependent on the processor architecture as it only initially uses resources on the CPU until main memory (RAM) is configured. Indeed, it is up to the firmware to initialize the main memory (which happens in the PEI stage under the UEFI spec). The code residing in this stage is generally designed to be as simple as possible, while the more advanced logic is handled later in the DXE stage.

The DXE stage loads what could be considered the user space UEFI environment. Driver interfaces are installed onto the initialized hardware to be used in the process of booting the operating system and during OS runtime. It is responsible for discovering, loading, and executing drivers in the correct order. Finally, the DXE stage passes control to the BDS where the OS boot loader takes over execution. A visual representation of this process can be seen online [41].

In the context of this project, pruning is performed on the modules executed in the PEI and DXE stages.

2.2 Firmware Layout

At a high level, UEFI firmware is composed of a flash descriptor region that identifies other regions in the image. This may include firmware for the Intel® Management Engine, or e.g. the network interfaces. The region of interest here is the BIOS region that follows afterwards.

The BIOS region space is split up into firmware volumes, each containing a collection of modules, Figure 1. Typically modules are grouped into a volume by their execution stage in UEFI. So, for example, one volume will contain the core start-up module for the DXE stage along with all of the other DXE modules to be executed.

A module contains one or more sections. Most importantly, some modules, but not all, contain a PE32 binary section that will be executed by the system at runtime.

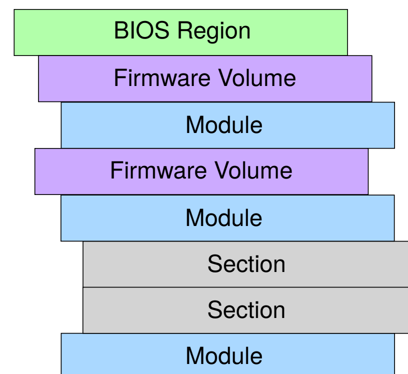

Figure 1: UEFI BIOS region layout

This project aims to exploit the modular nature of UEFI firmware in order to reduce the attack surface area of all motherboards that conform to the UEFI specification. Individual modules can be removed, with the BIOS region and firmware volumes rebuilt into a new, pruned image.

In 2017, Intel made a statement that they would be ending support for legacy BIOS compatibility by 2020 [23]. With manufacturers abandoning older proprietary legacy BIOS, this approach will continue to be valid for new motherboards.

2.3 Modules and Dependencies

For executable UEFI modules, one of the sections will contain a PE32 binary image. This is a standalone executable that is dispatched by the firmware. Executable modules will also contain a dependency (DEPEX) section, which will determine the order in which the modules are executed. During execution, modules will install pointers to functions using UEFI system functions. The installed functions are called protocols and are identified by Globally Unique Identifiers (GUIDs). Other modules use these GUIDs to look up the installed protocols and call into them. This is how standalone modules inter-link.

Each module has a DEPEX section that tells the DXE dispatcher what modules and protocols need to be initialized prior to executing it. If the DEPEX expression evaluates to true (i.e., required modules and protocols have already been loaded), the module can be loaded, otherwise it is postponed.

Unfortunately, the dependency section is not very helpful in determining which modules actually depend on one another. Protocols may be listed in the dependency section strictly to

change the dispatching order, not because the binary actually looks up the protocol and uses it. Likewise, protocols used by a module do not need to be listed in the DEPEX section if the protocol will already be installed by the time the module runs. A module may also have a soft dependency where it looks up a protocol, but still performs some valid behavior even if it is not present. The DEPEX section may be omitted entirely, in which case the module can be loaded right away. What is more, dependencies can be changed at runtime (when the DEPEX expression is evaluated), depending on various events in the environment. In short, the dependency section is only a reliable source of information for dispatch order, not for determining actual dependencies between modules.

There has been some work in reverse engineering these dependency lookups, but in a somewhat limited fashion. The method in [8] involved setting up a fake UEFI environment and then executing individual modules within that environment. Unfortunately, this does not fully account for system state when the modules are loaded, and modules that interact directly with hardware will not function properly. The only way to fully identify these dependencies would be to monitor the installation and lookup of protocols in the context of the real system. We detail this approach and explain our implementation of it in Section 4.3.

3 Pruning Strategy

3.1 Considerations

The selection of a pruning strategy should have two primary concerns: its runtime and the quality of the results it produces.

The property of a particular pruning strategy that most affects runtime is the number of test iterations that must be performed. The time required to perform a single test of a particular pruned state is on the order of minutes, so exhaustive searches simply aren't feasible.

As for quality, the number of modules removed is the metric most directly affected by choice of strategy; any strategy will remove one module at a time, and the order in which modules are removed determines how many modules are kept, due to the nature of inter-module dependencies. Therefore, the primary metric considered when comparing the results of different strategies is the number of modules removed. In Section 3.3, we discuss how other metrics, such as final image size and boot time can be incorporated as search heuristics, and in some cases may even lead to a reduction in runtime.

In Section 3.2, we present a few different representations of the search problem, considering factors such as module inter-dependency and the percentage of modules that can successfully be removed from the firmware. We then compare the average number of attempts performed and modules removed by a few natural pruning strategies and use the results to design a suitable pruning workflow.

3.2 Comparison of Existing Strategies

Assuming each trial takes a constant amount of time, the performance of any pruning strategy is proportional to the number of tests that must be performed.

One could consider subset-based reduction approaches like those used in delta debugging [46]. Delta debugging is typically used to find bugs rather than minimize software, however the principle is applicable to minimization. Delta debugging works by finding the "deltas"–lines changed, functions added/removed, etc.–between a program that passes a test and one that fails. The deltas are then recursively divided into subsets and tested in order to find a minimal set of deltas required to get the failing program to pass. In the context of DECAF, the passing program would be the original firmware image, the failing program would be an empty firmware image, and the deltas would be the UEFI modules.

However, these approaches rely on spatial coherence in the input, which in this case is a set of files in the firmware volume whose order have no real correlation to their removability. Delta debugging works best on well-structured inputs, and most approaches that utilize it rely on improving the coherency of the structure through high-level analysis [28] [40].

Another natural approach is to use a hill-climbing type algorithm that seeks to incrementally improve an existing solution by removing more modules and backtracking on failures. Hill-climbing can easily be used to incrementally improve the results of other strategies.

Another approach that will be considered as a baseline is to incrementally build a removal set R, initially empty. We consider one module m at a time, and if $m+R$ can be removed, we add m to R. We call this strategy linear removal.

As discussed in Section 2.3, some UEFI modules depend on others. The dependency graphs are Directed Acyclic Graphs (DAGs). The structure of the graphs themselves is not very interesting; they are simply very dense graphs. A few modules are referenced by nearly all others, and a few have no edges. However, the presence of these dependencies affects the runtime and removal level of the previously described strategies differently.

Consider Figure 2 where the dependency connectivity q is varied. q refers to a number of DAG edges to be selected randomly between the p removable modules. Assuming that roughly 60% of the firmware modules are removable, it can be observed that as expected, hill-climbing is able to fully prune the firmware regardless of the module connectivity, and the performance of the linear removal and delta debugging approaches is inversely proportional to q.

In order to achieve similar levels of module removal, linear removal methods could take on one of two approaches. They could repeat until the dependency tree is fully unwound, raising the complexity on an order of magnitude relative to the height of the DAG, or they could perform a linear removal to remove obvious candidates, followed by hill-climbing to

clean up the rest of the removable tree.

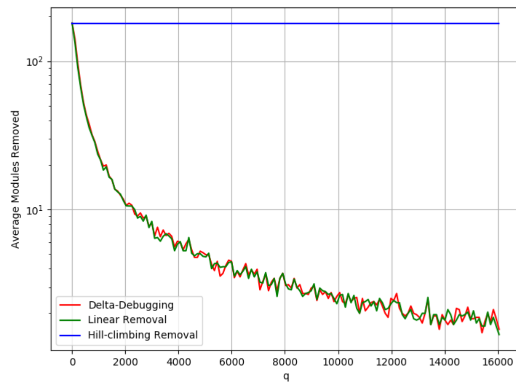

Figure 2: Average Number of Modules Removed with p=180 Modules Removable of n=300 Modules and Varying Dependency Connectivity (q)

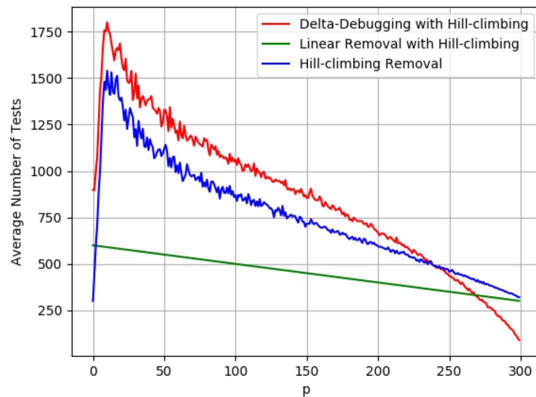

Figure 3: Average Number of Required Tests to Remove p of n=300 Modules with Connectivity q=25

Using an estimated value of $q = 25$ for the connectivity, a comparison of hill-climbing and linear removal with hill climbing methods can be seen in Figure 3. The linear removal with hill climbing is favored because repeatedly applying a linear removal approach results in repeated, redundant re-testing of modules that cannot be removed, while hill-climbing optimizes against re-selecting these modules.

3.3 Search Heuristics

Since exhaustive searches are infeasible, DECAF makes use of search heuristics: each module is assigned a weight that is updated throughout the runtime of the pipeline.

One can imagine a number of search heuristics that can be used to improve the runtime or results of a given pruning strategy. For example, if reduction of the overall image size is a primary goal, one can assign a higher removal chance to large firmware modules. If instead reducing the boot time of the final image is desirable, a module can be assigned a higher removal chance if removing it is observed to lower the boot time. This heuristic has the added benefit of reducing the time for a single trial, reducing overall runtime. Another potentially interesting heuristic would be one that runs some form of static analysis on the modules prior to pruning, giving a high removal chance to modules that are likely to contain some kind of bug or exploitable code.

One heuristic used to great effect in DECAF involves runtime UEFI module dependency. As described in 4.3, we inject two modules into the firmware image before the pruning process that report which modules install which protocols, and which modules subsequently look up those protocols during the boot process. This information can be useful in several ways. For example, a module with no dependencies may be assigned a high removal chance, while a module with many dependencies may receive a low one.

DECAF also halves the chance of a module being removed if a removal set including that module fails to pass the validation targets. The assumption is that modules that have failed previously are more likely to fail again. The intuition is as follows: a module can fail to be removed because (1) it directly provides functionality needed to boot the image or satisfy the validation targets or (2) its removal causes another module to fail, either preventing the image from booting or producing different validation results. If a module fails because it meets criteria (1), it will always fail. The potential for a module to fail because of reason (2) is mitigated by the dependency analysis and unwinding discussed Section 4.3.

3.4 The DECAF Pruning Strategy

DECAF deploys a single linear pass followed by a few rounds of hill-climbing, as it produces the best performance for firmware that roughly conforms to the model in which modules are either: removable, not removable, or removable if all of their dependencies are removed.

The workflow is aimed at finding a minimal image that passes validation targets. This is done by iterating across configurations of the search space until no further changes can be made (any change would cause validation tests to fail). The first iteration of the pipeline, performed on the vanilla image (empty removal set) will perform several extra steps:

1. Determine board manufacturer and configure various parameters (MAC/IP addresses, login credentials, etc)
2. Boot into an OS with the unmodified image and determine the hardware configuration (initial run for validation component).
3. Inject the dependency discovery modules (further described in Section 4.3) and generate the dependency graph based on runtime analysis.

After these tasks are completed, the pruning process can start. Having the dependency information, the removal probabilities are initialized. Initially, all modules are equally likely to be selected, excepting those that are present in the dependency graph. Modules that are part of the dependency graph have a smaller initial removal chance than the rest. The set of modules is then split in half recursively until the set contains only one module, at which point module removal is attempted.

Every iteration involves flashing the image to the motherboard, powering the motherboard, waiting for the OS to boot, and running the validation targets. If, at any point, a failure is encountered, the corresponding module's chance of being removed again is decreased by half.

After the modules are tried individually, the results are merged in the following fashion: if only one module set was removed successfully, return that set. If both succeeded, attempt to remove the union of the sets. If the removal succeeds, return the union of the sets. If the removal fails, return the larger of the two sets. The total number of removal attempts is the geometric sum $N + \frac{N}{2} + \frac{N}{4} + ... = 2N$.

The returned modules are then used as the initial solution for an incremental high-climbing approach to further improve the result. Modules are selected for removal based on a weighted random approach, using the weights calculated from the module dependency and failure information. This weighted approach is important because of the nature of dependencies between UEFI modules. A modified firmware image may fail because the removed module was a dependency of some other module, however that dependent module may not be essential. Further in the execution, the root of the dependency tree may be removed successfully, and as a result, all of the leaf modules can now also be removed. It is necessary to go back and retry modules that have failed because of this case. The weighting helps to ensure that less tested modules are more likely to be checked first while still preserving the option to retry previously failed modules.

4 Architecture and Software Stack

An overview of the architecture is in Figure 4. DECAF is composed of multiple modules, each responsible for a subtask of the overall pruning process.

DECAF needs to be capable of managing a physical board in order to control and monitor power, flash firmware images, and monitor overall hardware health. It needs to be able to prune firmware images and generate candidates to be tested during the reduction. These images need to be booted and validated in order to iteratively converge to a minimal image.

4.1 Workflow Engine

The Luigi [39] workflow engine (represented by A in Figure 4) was chosen for the high level management of the pruning process. The use of a workflow engine to manage the process serves a few purposes. It provides a high level task overview that can be used to monitor and manage the pipeline iterations. It also provides the ability to link tasks together with cached target data that is stored on the file system. This is a long-running process, which means that failures outside the scope of the pipeline may occur. A network or power outage are possible during this period and a recovery option is needed so that the progress is not lost. Because the workflow engine has the native function of caching its progress, the pruning process can simply be resumed at any point. Luigi's native concept of workers and dependencies also makes parallelization easy when multiple identical boards are available.

4.2 Firmware Pruning

A modified version of UEFITool lies at the core of the firmware pruning module. UEFITool is a mature UEFI firmware image editing application written in C++ with Qt. It is able to enumerate the contents of UEFI firmware as well as manipulate and insert modules and sections into firmware volumes. It works and is tested on a wide range of firmware across a variety of vendors.

We implemented a scriptable Python layer that utilizes the C++ backend of UEFITool, allowing for headless traversal and pruning of firmware images. This is a powerful tool (represented by B in Figure 4) for automating what was typically done meticulously by hand in UEFITool's user interface. The Python layer offers support for listing, inserting and removing modules while producing a structurally valid UEFI image.

4.3 Generating Firmware Dependency Graph

An analysis on the firmware image needs to be run in order to determine any dependency information. Our approach to identify these dependencies involves monitoring the protocol installations and look-ups in the context of the real system. Given the structure of an EFI image, modules can not only be pruned, but also appended to the binary.

DECAF appends two modules to the original image: (i) dependency probe, and (ii) dependency dump. The result is the "Dependency discovery image" in step 2 (Figure 4).

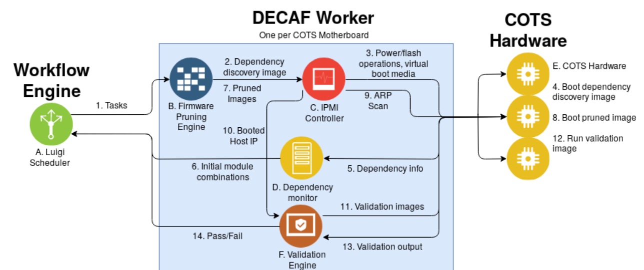

Figure 4: Overview of the DECAF platform architecture

Dependency probe is used to hijack several protocols that modules use frequently when interacting with each other (such as EFI_INSTALL_PROTOCOL_INTERFACE). The protocols are stored as function pointers in a structure that is passed to each module's main function. Overwriting these pointers very early in the DXE phase will cause all modules executing after this to use the hijacked functions instead. The hijacked protocols are simply wrappers over the original functions that also log the GUID of the calling module.

Collected data is stored in memory. Because the dependency probe is loaded at the earliest possible point in the boot sequence, right after the DXE Core, there is no way to transmit the information yet (serial/USB drivers/TCP stack are not loaded). Instead, the probe publishes its own custom communication protocol that exposes a pointer to the data.

The dependency dump module is loaded as late as possible, after the network stack has been initialized. At this point, most (if not all) module interaction has been recorded via the hijacked protocols. A look-up is necessary to find the information stored by the first probe. This information is then forwarded to an external server (represented by D in 4).

After the dependency discovery image is successfully booted and the data is collected (steps 4 and 5 from Figure 4), a directed graph is built from the module dependencies.

There are multiple approaches that can be taken at this point. Depending on the desired outcome, modules present in the graph can be excluded from the pruning process (this will result in a bigger final image, but it would attempt to preserve the original execution flow as recorded at runtime).

Another approach is to update the removal chance based on the degree of each node. All nodes found in the graph are less likely to be removed than modules that we have no information about (and were not recorded as active at runtime). Nodes with higher degree are less likely to be removed than those with smaller degree. The reason behind this it that a node with many incoming edges (or a module that is looked up and interacts with many other modules) is very likely to produce a failure if removed first, before the dependent nodes.

Figure 5 shows a zoomed in sample of a dependency graph.

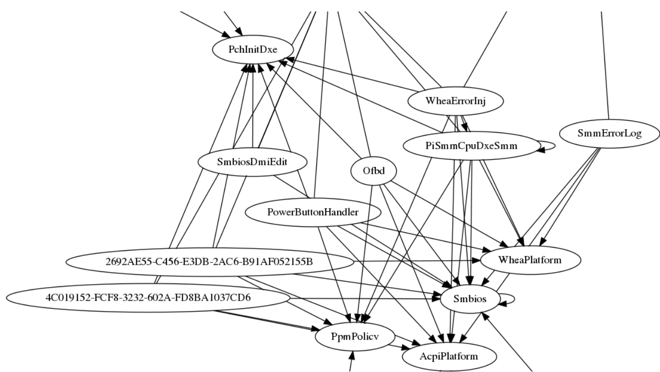

Figure 5: A sample dependency graph

Generally, there is a lot of inter-module interaction, and there are even some self-loops. This can represent a module that awaits an event in the environment, and periodically probes itself. Removing a module that is called by one or more of its peers will increase the chance of failure. A good strategy for the pruning process is to first remove modules that have no or only a few incoming edges (such as EventLogsSetupPage or Ofbd in Figure 5), and only afterwards attempt to remove nodes that are deeper in the graph.

In this particular case, the graph from Figure 5 is generated from the firmware of SuperMicro A1SAi-2550F. The original image contains 244 modules while the full graph has 147 nodes (modules) and 3881 edges (inter-module interaction). This leaves 97 modules that have no recorded interactions at runtime, but they are not necessarily unused: they may not interact with other modules, or they may only be called during very early initialization, before our hook is introduced. Out of the 147 recorded modules, 100 nodes have an in degree of 0 (i.e., no dependents), making them the second best removal candidates after the modules that have no data recorded. 21 modules have an in degree of 147. These modules are likely to contain core functionality as they interact with all others. Removing them will likely produce bad images. These statistics will of course vary for different firmware images.

Some modules are named, while others are represented by their associated GUID. Generally, named modules are well known and provide standard functionality (and are reused across models/vendors), while the others may be custom. For example CsmVideo adds graphic support for backwards compatibility with older BIOS features, while the Whea modules (Windows Hardware Error Architecture) provide error management and log information for the OS [35].

As the graph is generated before pruning, knowing module names and interactions can provide valuable information to the user looking to white/black list certain functionality.

4.4 Board Management

Testing changes to the UEFI firmware is not a goal that can be simply achieved using virtualization tools, such as QEMU [43], because QEMU does not really virtualize the hardware below the guest OS. The guest can only see a memory map where accessing particular addresses will result in various side effects (such as manipulating hardware via registers). QEMU replicates this behaviour, while mimicking the side effects the OS would normally see on dedicated machines. The UEFI environment itself, placed lower in the software stack, is more difficult to virtualize and QEMU does not support hardware profiles compatible with modern UEFI systems. Indeed, QEMU only supports two x86 chipsets: i440FX and Q35. Both are quite old (1996 and 2007, respectively), and there do not exist many (if any; we were unable to find one) compatible UEFI motherboards.

Open Virtual Machine Firmware (OVMF) is a project that

enables UEFI support in virtual machines [14]. It is based on the EDK II implementation of the standard, and we have used it for various tests and prototyping. But ultimately, the goal of DECAF is to work on a large number of COTS platforms, and the OVMF image provides only a limited and considerably different simulation of a real board. Taking this into account, the only way to test whether a pruned firmware image is functioning correctly is to flash it onto the motherboard and boot an operating system to validate that everything is still working as expected by running a test suite. This requires controlling the motherboard in an automated fashion to accomplish a few tasks: (1) power control, (2) power monitoring, (3) flashing firmware images, and (4) providing boot media.

For convenience, motherboards with a BMC (Board Management Controller) that provides IPMI (Intelligent Platform Management Interface) were selected for DECAF since they offer all of the services required. We developed a unified Python API (represented by C in Figure 4) for interfacing with the motherboard IPMI services, hiding vendor specific behavior. IPMI is typically only present on server-grade hardware, but the same thing can be accomplished on consumer hardware with an external flash programmer, a GPIO controller for monitoring and controlling the power, and physical or PXE boot media. When implemented behind the API, this would work seamlessly with the rest of the components.

Because the aim of DECAF is to harden trusted code base residing at the firmware level, it is worth mentioning that the various IPMI implementations are not really secure, as emphasized by [9]. This is consistent with some of our initial findings when developing the vendor specific extensions. Nevertheless, this does not represent a liability for our goals, as the pruning operation is a one time process and the resulting image can be flashed on boards that have the IPMI disabled. Also, as previously mentioned, the presence of IPMI is a convenience, not a necessity.

4.5 Validation

The first priority in validation is to make sure that a motherboard flashed with modified firmware actually manages to boot into an operating system (ArchLinux 2018-11-01 was used to produce the images described in this paper). On boards that support it, POST (power-on self test) codes are monitored through the board management API to monitor early execution. This is done as a time-saving measure. If a timeout is reached and the operating system has not booted, the firmware is considered broken, and the process backtracks and continues down a different pruning path. However, by monitoring the POST codes, it can sometimes be determined that a firmware image is broken without waiting for the entire duration of the timeout period. The whole pruning process tends to run over the course of a few days, so any time savings that can be obtained are valuable.

The IPMI controller monitors the network and waits for the Linux boot media to bring the motherboard's network interfaces up and negotiate DHCP. It then provides the IP address of the booted host to the validation engine (represented by F in Figure 4), which uses SSH to remotely access the operating system where it can perform tests. At the beginning of the pruning process, the stock firmware image is flashed and the validation component collects information from the known-good booted operating system. This is used as a baseline when comparing the collected data from the modified images. For example, the PCI hardware configuration of the image is recorded so that on subsequent tests it can be determined if any of the hardware components on the board were not brought up properly.

Once the operating system is up and SSH connection is established, any sort of tests can be performed. The validation component is meant to be flexible and extensible. We use docker to ensure portability and extensibility: each validation target is a docker container which is built at the beginning of the pipeline and copied to the booted OS over the network. The container is run and the output is compared to that of the baseline firmware. If there are any differences, the flashed firmware is considered invalid, so any tolerable differences must be filtered out by the container itself. For example, in the dmidecode validation target discussed later in this section, we check only memory and CPU configuration types. This is because dmidecode was specifically added to preserve memory timings and clock frequency early on in our data center pruning efforts. Other System Management BIOS (SMBIOS)/Desktop Management Interface (DMI) information (OEM strings, system configuration options, etc.) are not strictly necessary to the functionality of the device, but of course can be easily included if a user desires.

As will be discussed further in Section 5, the pruning pipeline was run with two profiles, "aggressive" and "data center." The functional difference here is the motherboards are booted off of virtual media provided through the IPMI interface in aggressive mode and over iPXE in data center mode. Therefore, iPXE and related components (e.g. network drivers) will be preserved in the data center pruning, while they may be removed in the aggressive pruning. Each profile uses the same set of validation targets, detailed below:

1. **dmidecode** is used to decode the DMI table, which is hardware configuration information reported by the firmware to inform the operating system of the hardware present in the system and facilitate management. This ensures important information such as configured memory speed is preserved.
2. **lspci** is used to validate that detailed information about PCI buses and related interactions is preserved.
3. **/proc/acpi** is checked to ensure the operating system will be able to perform ACPI power management.
4. **Intel's CHIPSEC** security suite is run to check the security of pruned images.

The security of the pruned firmware images is of utmost importance. With the goal of improving security by reducing the byte surface area, it must be ensured that removing certain modules does not introduce new known vulnerabilities into the firmware. For example, there may be a module responsible for write protecting the SPI flash chip containing the firmware, which prevents attackers that manage to infect the operating system from permanently taking over the hardware at a low level. Another may serve as a lock box, putting the S3 resume script into safe memory so that attackers cannot use it to penetrate the system [31].

Intel's CHIPSEC framework is used to monitor and validate the security integrity of these modified images [24]. CHIPSEC scans the system for known firmware level vulnerabilities and reports them; these reports are compared against the report from the original image to ensure that no additional vulnerabilities are introduced by the pruning process. Each vanilla image had a few failures, such as the SPI chip being writable or Spectre/Meltdown style attacks being possible. Further, e.g., our HP server contains four critical errors: one stemming from Spectre-style vulnerabilities, and three from improperly configured protections that may allow an attacker to modify the bootflow, overwrite SMRAM via Direct Memory Access (DMA) attacks, or even overwrite the BIOS through the SPI chip.

DECAF prunes modules but does not (yet) patch modules (i.e., to fix such vulnerabilities in remaining modules). As a result *the CHIPSEC vulnerabilities cannot be fixed automatically* by DECAF.

Any additional protections can be added manually [38]. In future releases, DECAF may automatically handle this.

If DECAF is being run with a certain objective in mind, tests can be specifically crafted in a manner that assures the desired functionality is preserved. This guarantees that the user's needs are satisfied, while potentially increasing the number of modules pruned.

Indeed, one can imagine any number of tests that may be considered essential to a certain application. If more complex tests need to be run, it is possible that the time required to validate a single pruning profile may increase substantially (e.g., if some sort of stress/performance test needs to be performed). The initial use case for DECAF was for hardware running in cloud data centers for compute-as-a-service where features such as USB support, VGA support, etc., are not necessary, and thus validation can be performed rather quickly.

Certain hardware features, while present, may not be required for a user's application, allowing for even greater pruning. There are two methods for achieving this. First, if the user has prior knowledge on what modules are responsible for the functionality that is no longer needed, the modules can be removed from the start via the blacklist. If this is not the case, the user can make sure that the validation layer ignores the respective feature (e.g., ignore that the device associated with the serial port is no longer listed in the OS).

5 Results

The pruning process was run with two profiles: "aggressive" pruning, where only booting from physical media (or physical media emulated by the board's BMC) was required, and "data center" pruning, where the boards were pruned for the purpose of running in cloud data centers offering compute-as-a-service, booting over iPXE.

A visualization of the aggressive pruning process can be seen in Figures 6 and 7 on firmware from two different motherboards: the SuperMicro A1SAi-2550F and the Tyan 5533V101, respectively. Here, the markings indicate the result of attempting to prune the board, with blue (BIOS Post) indicating that the firmware did not boot, red (OS Probe Failure) indicating that one or more of the validation targets failed, and green (OS Probe Success) indicating that the validation targets passed. The SuperMicro board is based on an Intel®Atom C2000™ chipset, and the Tyan board was based on an Intel®Core i3™ Haswell chipset.

The results of the aggressive pruning pipeline and the data center pruning pipeline can be seen in Tables 1 through 4. The aggressive pipeline was able to remove a much larger portion of the firmware than the data center pipeline, removing over 70% of the firmware bytes from the SuperMicro motherboard and almost 40% from the Tyan and HP motherboards. The pruned image boots more quickly as well. The SuperMicro motherboard booted 13 seconds faster on average, and the Tyan motherboard booted 7 seconds faster on average with the pruned firmware.

Data Center. One major DECAF application has been to prune images for a cloud data center. The Tyan 5533V101, the SuperMicro A1SAi-2550F, and other models have been successfully used as part of an OpenStack deployment, in a production data center successfully since 2017, with performance and reliability metrics higher than standard firmware across hundreds of thousands of instance allocations. For data center pruning, the results are also, strong, ranging from about 7% to about 30%. More recent results suggest this figure is closer to 40% (e.g., on the HP motherboards).

Security metrics are evaluated later in this section.

5.1 Comparison Between EFI Images

Testing with a large number of boards and vendors has proven difficult. The IPMI based communication is not necessarily standard (nor too well documented) for each vendor. This means that the API exposed by the IPMI is different, and the submodule of the project that deals with this needs to be adjusted for each vendor accordingly. Secondly, virtualization does not produce good results: the virtualized environment is highly different from a real board in terms of BIOS: the modules loaded are different and the hardware emulated is different (and not customizable enough for our purposes).

Because of this, a different testing direction was taken: an-

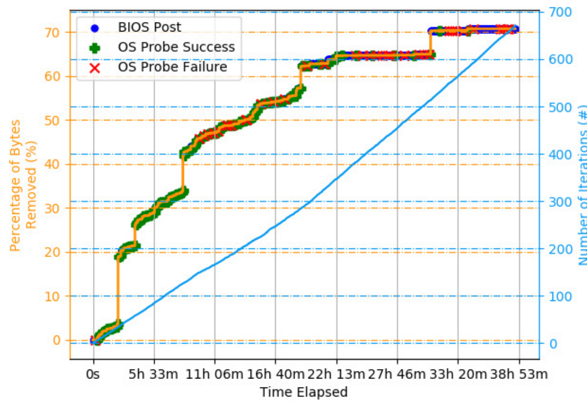

Figure 6: Percentage of bytes removed and number of iterations over time for SuperMicro A1SAi-2550F firmware

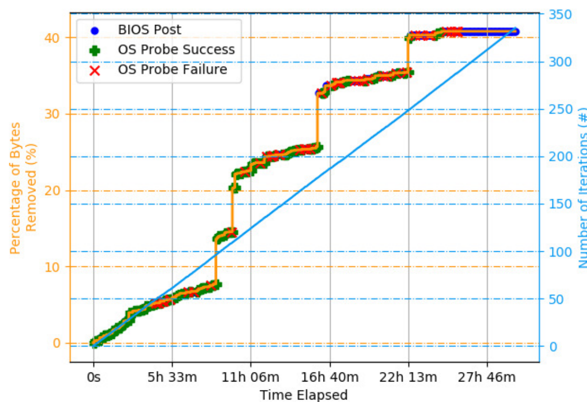

Figure 7: Percentage of bytes removed and number of iterations over time for Tyan 5533V101 firmware

alyze just the binary images from a number of vendors and assess their similarities. It is valuable to determine to what degree these images are overlapping. Taking into consideration the structure of a UEFI image, it is convenient to compare the number of modules that are present in multiple versions, from different vendors. There is no direct and unbiased bitwise comparison method for binary images, as often enough there will be areas padded with 0 (or other characters, various encodings, etc). Also, without having access to the source code of the firmware images, bitwise comparison is made even more difficult by the compilation process: different optimization levels and architectures will result in vastly different binaries, even if the code base is identical, or highly similar.

Instead we take advantage of the GUID. While an EFI module is not necessarily uniquely identified by a GUID, we can argue that the base functionality between modules with the same identifier is largely the same. A GUID is a 128 bit random generated quantity that is uniquely associated to each module, and is aimed to work similarly to a hash, according to [1]. For more information on GUIDs see [2].

Two images can be compared by extracting the list of GUIDs present in each, and determining the common ones. It is important to note that all motherboards have inherently similar functionality, and their firmware is based on a common open source implementation. This aspect will cause a rather high overlap rate in images, even from different vendors. We are interested in how high the match rate is, and if it supports the claim that firmware is being mass produced and bloated.

In order to keep the comparison unbiased, the motherboards models were chosen at random. Some are for desktops, some for laptops. There was no prior knowledge about their functionality and possible similarities.

Our case study was done with three different scenarios in mind. First, we compared the similarities between 5 randomly chosen EFI images, from 5 different vendors. Second, we wanted to explore the usage of the same modules within 5 different EFI images that were created by the same vendor. Lastly, we took a closer look at how often UEFI firmware updates actually change modules present in a given image.

Table 6 shows a comparison of 5 different products, picked from various vendors. The first number represents the modules in common, and the second value represents the percent of common modules between the two images, with respect to the larger image.

For example, 257/26% tells us that the Asus and the ASRock motherboards have 257 modules in common, or 26% of the bigger image (ASRock) is found in the smaller one (Asus). As we can observe there are several cases where the smaller image is over 50% identical with the larger one.

Similarly, Table 7 contains a comparison of 7 of the most popular motherboards from ASRock. The boards were chosen from different product lines, and firmware images from the same series are almost identical. It can be observed that these motherboards have a rather large number of EFI modules on average (up to 900 in some cases). This causes an even bigger similarity between the binary images. Given the sizable number of modules, out of which many are overlapping, it is probable that after the pruning process, a substantial decrease in the image size would be obtained.

Table 8 contains a comparison between the patch versions of the same model (ASRock IMB186 motherboard). As expected, these patches produce very little change from version to version. We can observe that the original 257 modules were propagated until the current version (v2.3). Also there is a 100% match between several versions (this happens because the changes are below modular granularity).

The data collected indicates a considerable percent of code is being reused across various modules, as initially asserted. We can observe that in some cases up to 70% of a firmware image is found on a different model from a different vendor (see Table 6, Asus vs ASRock). Furthermore, between the models of the same vendor, the matching percent can go up to 100% (having 2 different motherboards run very

similar firmware). There is almost no difference between different patch versions of the same model (generally a few new modules added). Given the large amount of overlap between different UEFI firmwares, it is easy to see why a vulnerability found in a single firmware may be reproducible across a wide variety of mass-produced hardware (as discussed in [45]).

5.2 Benefits of Reduced Vulnerability Surface

Benefits of code reduction include: reduced TCB – at the industry-average of 1.5-5% bugs per line of code [27], this can add up to thousands of (undiscovered) bugs and hundreds of exploits – reduced boot time, the ability to fit the firmware onto a smaller SPI chip etc, removal of physical attack vectors such as over peripherals (e.g., USB), and a reduction in the number of Return Oriented Programming (ROP) gadgets etc. In this section, we provide an analysis of these benefits.

5.2.1 Industry Standard BPLOC Metrics

The number of bytes generated from one line of pre-processed C code by an optimized compiler has been estimated [16] at around 14. This allows an estimation of the number of source code lines used to produce the firmware images. We can then calculate the number of lines removed using the reduction in byte surface area (Table 4).

The industry-average number of bugs per line of code (BPLOC) [27] has been estimated as 1.5-5%. This allows an estimation of the number of undiscovered, removed defects for different motherboards. Under aggressive pruning, an estimated 2261 bugs were removed from the SuperMicro A1SAi, and 2791 from the Tyan. Under data center pruning, the number is as high as 1005 (Table 5).

5.2.2 Removing Infrequently Used Features

Further, removing rarely used features (features likely to be removed by the DECAF pipeline) provides a proportionally higher benefit. Rarely used features are more likely to contain errors, since the resulting bugs are less likely to be discovered and therefore less likely to receive development attention beyond in-house testing [22].

5.2.3 Pruned Code is not Unreachable

At first glance, it may seem that any code whose removal does not affect functionality is unreachable. This is not the case with a vast majority of DECAF-pruned modules and can be validated by the significant reduction in boot time which shows modules are part of the control flow.

Further, there are numerous vectors by which an attacker indirectly gains access to code that is not entirely run in a standard boot sequence. For example, consider a firmware image that contains two DHCP modules: one from the EDK II standard and one from the manufacturer. Suppose the EDK II module contains an exploit, but the manufacturer module is loaded by default. If an attacker can cause the manufacturer module to fail to execute (perhaps because it contains a less significant defect), then the EDK II module will be loaded when another module looks up the DHCP protocol.

Similarly, ROP gadgets can be used to load a normally unused module directly (Section 5.3).

Finally, consider the case of a driver for an obsolete peripheral. This code may not execute during a normal boot sequence, but may be executed if the booted operating system requests such a driver. If the module contains a serious exploit, an attacker that gains control of the operating system can cause the driver to be executed, escalating an operating system attack to a firmware attack. This could pose a serious permission escalation if, for example, the hardware owner's intention was to prevent the OS from accessing the firmware payload on the SPI chip (e.g., for bare-metal cloud).

5.3 Mitigating Existing Attacks

Finding and directly patching existing known firmware bugs is not within the scope of this work. Indeed as noted in Section 4.5, no CHIPSEC reported bugs disappeared after pruning. The goal of DECAF is to maximally reduce the vulnerability surface of the hundreds of bugs that are still unknown.

In fact firmware vulnerabilities (some fatal [38]) do not receive anywhere near as much attention, publicity, and tracking when compared with OS and software vulnerabilities. A search for "UEFI" reveals only 23 results in the CVE database [6], many of which are related to a single USB issue. Searches for specific models or product lines we pruned reveal a few more, but virtually all relate to the BMC and not the firmware itself.

Nevertheless, in addition to reducing the overall vulnerability surface, DECAF also helps mitigate a number of common attack vectors including: Return Oriented Programming (ROP), USB attacks, SMM attacks, and network attacks.

5.3.1 Return Oriented Programming (ROP)

ROP allows an attacker to hijack the control flow of a program by executing a specific set of instructions that are already found within the original code. This type of attack is based on gadgets (short sequences of instructions followed by a return) assembled together through stack-originated calls. There are two similar classes of attacks, Call Oriented Programming (COP) and Jump Oriented Programming (JOP). These are similar to ROP but make use of call and jump instructions, respectively. Attacks start with a buffer overflow hijacking the control flow, e.g., by sending malformed network packets processed by a faulty UEFI driver.

Using the buffer overflow, a function pointer or some part of the executable memory is overwritten with a malicious sequence. By manipulating the stack, the attacker can then

jump into a gadget, and each gadget indirectly branches to another, allowing execution of arbitrary code, subverting the original control flow of the application.

Crucially, gadget-style attacks are (sometimes exponentially) easier with increasing code base. A single gadget may modify the control flow or program memory in a limited way. However, chained gadgets can be made Turing complete [34].

Firmware contains large numbers of potentially exploitable gadgets. However, not all gadgets are equal in terms of usefulness when mounting an attack. Gality [4] is a tool that seeks to analyze the entire set of gadgets available in a binary and determine how many of them are "high-quality." Table 2 and Table 3 illustrate the numbers obtained using this tool on several firmware images. DECAF pruning reduces the total gadgets available by 12 to 64% and reduces the high-quality gadgets available by 11 to 62%. Gadget quality is evaluated based on type (arithmetic, logic, control flow, etc), pre-conditions and side effects on the stack, and whether popular known attacks are possible with the given gadget collection [15].

5.3.2 USB Attacks

Another extremely common attack vector is a motherboard's USB port. There are many known USB attacks, many requiring no further user interaction than plugging in the device, and some are even able to re-flash the firmware [30]. For data center scenarios, DECAF routinely prunes USB and other unnecessary peripherals, completely eliminating the attack vector (Section 5.4).

5.3.3 SMM Attacks

System Management Mode (SMM) is a privileged execution mode. During the DXE phase, System Management Interrupt (SMI) handlers are loaded into SMRAM. When an SMI is trigged, the handler runs in this highly privileged state. The handlers can communicate with the operating system through a shared buffer. This presents two new attack vectors: 1) if an attacker can overwrite SMRAM, she can execute arbitrary code in a highly privileged state, and 2) if she can gain access to the SMM communication buffer, and there exists an exploit in an SMI handler, she can escalate an OS attack into a BIOS attack.

Kallenberg et al. [20] construct an attack of the latter type. The firmware in question (Dell Latitude E6400, BIOS revision A29) provides an SMI routine that allows flashing of the BIOS from the OS. The routine reads packets from the SMM communication buffer, reconstructs the BIOS update image, and verifies its integrity. However, a flaw in the packet handling allows for a stack smashing style attack, which the authors show can be used to flash a malicious, unsigned BIOS image. DECAF can (likely automatically) prune the module that installs the BIOS update SMI routine. The BIOS menu can be kept, thus removing this exploit vector while still allowing BIOS updates from the BIOS itself.

The above exploit is CVE-2013-3582 [19]. A search for SMM related CVEs [5] reveals 24 other potential applications for DECAF. However, many are self-disclosed (e.g., by HP and others etc) and do not provide attack details.

5.3.4 Network Attacks

Other important attack vectors center around the (sometimes multiple) network stacks present in the firmware. The network stack is needed by services such as DHCP, FTP, and PXE in the pre-boot environment. Simple attacks include, for example, exploiting the lack of signatures and authentication in certain DHCP servers: preempting a legitimate DHCP server, and inducing the BIOS to boot a malicious image and take over the existing operating system. This has been demonstrated by Matt Weeks at Defcon 19 [42]. DECAF prunes any unnecessary network stacks and can also be used to remove associated services (e.g., DHCP) to thus completely remove an attack vector often exposed by sysadmin negligence.

5.4 Feature-Specific Pruning

While the primary use-case of DECAF is to produce the most efficient, minimal images retaining a desired set of functionality, it can also be used to instead remove one or more desired features while retaining as much of the original image as possible. For example, some features may not be desirable on certain critical hardware; removing USB or GPIO support in order to prevent physical access to a device is a common scenario in security sensitive contexts. Another example would be disabling unused hardware components to save power.

For this approach a goal can be set for a maximal image that will behave like the original with the exception of the one removed feature. To this end, DECAF runs up to the point where the target feature is pruned. After this, the process is reversed and modules are inserted back incrementally until the original image is as close to the original as possible, while still missing the target feature. Inter-module dependencies still represent a constraint here and this is the reason why DECAF cannot simply add everything back after the target feature is disabled. It is important to note that there is no guarantee the target feature can be disabled by removing a single module from the image; a set of modules might be removed in order to achieve the desired effect.

Further, to disable the support for a given feature, other side effects may appear – e.g., DECAF may not be able to remove a single USB port; only all USB ports.

As an example, consider the SuperMicro A1SAi-2550F motherboard. Pruning to eliminate USB support results in a removal of 6 modules out of a total of 244.

6 Discussion

6.1 Limits of BPLOC as a security metric

Industry-average BPLOC (bugs per line of code) [27] as a security metric has obvious limitations.

Primarily, it does not really address or represent any existing known vulnerability. No CVE entry will be related to generally reducing vulnerability surface.

Secondarily, psychologically it is easy to overlook and posit that if only developers are more careful, this rate will go down. Yet, unfortunately this is not true. Even extremely rigorous processes such as put in place by Microsoft still yield "about 10 - 20 defects per 1000 lines of code [KLOC] during in-house testing, and 0.5 defect per KLOC in released product".

Thirdly, not all of the 1.5-5.0 average bugs introduced for every hundred lines of code can be turned into viable exploits. Yet, even if only 1% of them do, this results in tens of zero day vulnerabilities for even the simplest firmware we tested.

6.2 Limits ROP as a security metric

Using ROP as a security metric in previous works has garnered some criticism. Crucially, ROP gadgets are almost never eliminated entirely, and therefore the benefit of reducing their count is reduced by the fact that the remaining gadgets may still provide viable exploit paths.

[10] shows that in the case of the source code trimming tools CHISEL and TRIMMER tools, debloating can in fact introduce new gadgets, including some that are even more exploitable than what existed previously.

Note, however, that this is only true of intra-source code trimming techniques, which may result in wildly different instructions in the final binary. Since DECAF prunes entire, self-contained binaries, it does not rewrite code nor does it rearrange the existing control graph in binary blobs, and thus introduces zero new gadgets. This also means that the reduced gadget count really represents the removal of entire attack vectors. Removing a module with high quality gadgets means none of those gadgets can be used to craft an exploit.

Nevertheless, existing ROP-reduction related criticism still holds: as long as *some* gadgets are left, ROP may still be feasible albeit in a more limited form.

6.3 Limitations of Validation

There are, of course, limitations to automated removal. For example, only the functionality required by the validation tests is guaranteed to be preserved and special edge cases may be challenging to handle. For example, a module may depend on other error handling modules only in the case of hardware errors (which are not triggered or emulated during pruning). Pruning the error handling modules may result in undefined behaviour. This hypothetical may require special handling, however we note that no such examples can be found in the core EDK II codebase.

Overall, 100% test case coverage for outlier scenarios is obviously not feasible. This is why special care must be taken to ensure that the validation targets match the intended use cases of a particular pruned firmware. For example, if the firmware is intended to be used in a NAS box, validation targets will test RAID functionality, read/write speeds, and (simulate) hardware (e.g., disk I/O) failures. Indeed, the validation requirements are simplest (and the pruning potential greatest) where limited functionality is required, such as our aggressive profile or Data Center pruning (Section 5).

Finally, we note that BIOS functionality is to be minimal anyway. Apart from driving highly esoteric motherboard-specific hardware (which would likely employ non-UEFI firmware anyway), most functionality is often taken over by OS drivers which are more powerful and up to date.

In our experience of successfully running heavily pruned images in production data centers since 2017, having the OS successfully boot and pass basic sanity checks is sufficient for thousands of even the most demanding enterprise applications running on top.

7 Future Work

In ongoing work, DECAF is being augmented to perform static analysis and binary module payload reduction on individual modules. We'll use existing work [7] as well as newly designed mechanisms for symbolic execution to further optimize pruning.

In addition to analyzing and pruning at sub-module level, DECAF would be greatly enhanced by the ability to patch modules to enable certain platform protections where they are missing, such as the ones described in Section 4.5.

Expanding and perfecting our set of validation targets is something we are continuing to work on. One validation target that we experimented with was the Firmware Test Suite (fwts).

fwts [21] is a comprehensive set of tests of operating system/firmware interactions. It executes 113 test suites that include all CHIPSEC tests, ACPI, error reporting mechanisms, CPU and memory states, and hand-off to the main OS. We were able to achieve similar pruning percentages without any degradation on the test results using only fwts.

To mitigate long running validations, the pipeline will be extended to allow specific validation targets to run only in certain cases (e.g., after a certain pruning size etc), backtracking to the last passing profile if the target fails. This allows for longer-running validation targets to be included without dramatically increasing the overall pipeline runtime.

8 Related Work

Program slicing allows programmers to obtain the minimal software form that provides a particular behavior [44]. This approach is typically used for specific purposes such as testing, debugging, compiler optimization, or software customization. The reduction of a program can be done either statically, e.g., by determining the Control Flow Graph and removing unused nodes, or dynamically, e.g., by decomposing the program execution, typically while debugging, and identifying only statements/variables of interest.

Debloating software is a mechanism that focuses on determining the unused code of a program and removes it. Modern compilers already implement functions to eliminate dead code through static analysis, hence, most recent work focuses on dynamic elimination. Heo et al. present a novel approach to program debloating using reinforcement learning [17]. In their work, they present motivating examples wherein static analysis and dynamic analysis alone cannot remove all the dead code and security vulnerabilities in the code.

Both program slicing and debloating software mechanisms can be used to improve our pruning mechanism, however there are two important aspects to be considered before one can adopt and adapt them. First, existing research focuses on trimming a self-contained program that can be run independently of other system components, while UEFI firmware initializes system hardware. An error may prevent the operating system from using some hardware features, but the UEFI firmware itself will still continue to run without problem. Second, the problem of hand-written assembly code in UEFI firmware is not tackled by most of the existing literature. The EDK II project contains about 1.4M lines of C/C++/Header code and 19K lines of assembly, a small but not insignificant amount.

Rastogi et al. use dynamic analysis techniques to automatically debloat and harden docker containers, removing unused resources and partitioning the executeables within the container based on the resources they access [33]. They use system call logs to determine resource access which is similar to our approach of hooking into the UEFI protocol look-up method discussed in Section 2.3.

Bazhaniuk et al. use symbolic execution to find vulnerabilities within UEFI firmware by analyzing a snapshot of SMRAM [7] . Their setup can generate 4000 test cases in 4 hours, which can be later repeated on an actual real board. Their testing environment makes use of a generic and open source UEFI implementation, and replicating it on a closed source UEFI might not be possible, given the difficulty in emulating non-generic hardware.

The article from [22] presents an extremely similar approach, but focused on debloating the Linux Kernel instead. In this case, the argument made shows that the kernel will contain a very large set of features, out of which only a small number will be used by a specific end user. The developers include all available functionalities in the kernel, even if support for certain exotic features is used by only a few users. In a similar manner to our work, a set of usage scenarios are defined in order to determine what parts of the code are reached within the targeted kernel. This is achieved by analyzing the function call graph at runtime during a use case. The functions are traced back to the source code, allowing the creation of a custom configuration. According to this work the Linux Kernel has roughly 11,000 configuration options, which will be automatically tailored to minimize the code base while maintaining the functionality determined in the usage scenarios, removing up to 70% of it.

RedDroid [18] is a project that targets software bloat in the Android world. Here redundancy is defined as either compile-time or install-time, depending on when it can be determined. The first category comes from included libraries (because each application runs inside a Java Virtual Machine, there is no static or dynamic linking). The second one refers to various platform dependent files (which can only be determined as redundant when installing on a specific platform). The software debloating is realized by static code analysis (for compile-time redundancy; reachable code is determined, removing the rest) and a set of shell scripts (for install-time redundancy; the scripts will remove any unnecessary platform specific files). On average the APK size can decrease by 42%. It is important to note that RedDroid does not necessarily focus on security, but rather on saving hardware resources.

The work at [13] presents a large scale experiment on embedded firmware images (note: in this context firmware does not necessarily mean UEFI environments but, rather any form of software that may be found on various embedded/IoT devices). A large number of binaries was collected (roughly 32000 through web crawling). These images were processed using simple static analysis and correlation techniques. By comparing various binaries, known vulnerabilities were be detected on various devices that were previously not known to be affected. 38 new CVEs were also submitted, as the framework also attempts to extract and crack password hashes, private keys and certificates, find back doors and target various other common hot spots. An interesting result is that two different classes of products had the same vulnerability (44 surveillance camera models and 3 firmware images for home routers). It turns out that they all used a System on a Chip (SoC) for networking devices from the same vendor. This particular scenario shows how vulnerable software is reused in different applications, and a pruning framework (such as DECAF) can potentially remove such threats.

Of particular interest to many security-conscious users is the Intel Management Engine (ME), which is co-processor integrated into almost all Intel-based motherboards since 2006. It enables many Intel Features which may be attractive to some enterprise users, but requires full access to the host system's memory to do so. For users not needing the advanced management features, the ME is simply another poorly understood attack vector. Multiple vulnerabilities have been

identified in the Intel ME in the past, including CVE-2017-5689 [26], which can give an attacker full access to the host system, including installing persistent malware and modifying firmware. The open source project me_cleaner [12] contains scripts for patching the ME firmware to disable it on a wide variety of motherboards. me_cleaner, in conjunction with patching and removing parts of the UEFI BIOS that depend on the ME, was used to disable Intel ME in certain SuperMicro boards used in cloud data centers.

9 Conclusions

DECAF is the first extensible modular platform capable of automatically pruning a wide class of commercial, off-the-shelf UEFI motherboard firmware, in some cases by over 70%, significantly limiting attack surface areas and hardening the resulting stack. DECAF is available freely for the research community to use.

References

[1] GUID FAQ. https://github.com/tianocore/tianocore.github.io/wiki/GUID-FAQ.

[2] RFC 4122. https://tools.ietf.org/html/rfc4122.

[3] EDK II Project, https://github.com/tianocore/edk2, 2019. https://github.com/tianocore/edk2.

[4] Gality - open-source implementation to compute metrics on sets of gadgets, Nov 2019. https://github.com/michaelbrownuc/gality.

[5] Cve results for "smm", Apr 2020. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=smm.

[6] Cve results for "uefi", Feb 2020. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=uefi.

[7] Oleksandr Bazhaniuk, John Loucaides, Lee Rosenbaum, Mark R. Tuttle, and Vincent Zimmer. Symbolic execution for bios security. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., 2015. https://www.usenix.org/conference/woot15/workshop-program/presentation/bazhaniuk.

[8] Jethro Beekman. Reverse engineering UEFI by execution. *32nd Chaos Communication Congress*, page 20, December 2015.

[9] Anthony Bonkoski, Russ Bielawski, and J. Alex Halderman. Illuminating the security issues surrounding lights-out server management. In *Presented as part of the 7th USENIX Workshop on Offensive Technologies*, Washington, D.C., 2013. USENIX. https://www.usenix.org/conference/woot13/workshop-program/presentation/Bonkoski.

[10] Michael D Brown and Santosh Pande. Pdf, Feb 2019.

[11] Yuriy Bulygin, Andrew Furtak, and Oleksandr Bazhaniuk. A tale of one software bypass of windows 8 secure boot. 2013.

[12] Nicola Corna. Me cleaner, Oct 2018. https://github.com/corna/me_cleaner.

[13] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 95–110, San Diego, CA, 2014. USENIX Association. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/costin.

[14] Laszlo Ersek. Open virtual machine firmware (ovmf) status report. Technical report, Red Hat Software, July 2014. http://www.linux-kvm.org/downloads/lersek/ovmf-whitepaper-c770f8c.txt.

[15] Andreas Follner, Alexandre Bartel, and Eric Bodden. Analyzing the gadgets - towards a metric to measure gadget quality. In *Proceedings of the International Symposium on Engineering Secure Software and Systems*, 2016.

[16] Les Hatton. Estimating source lines of code from object code: Windows and embedded control systems, Aug 2005. http://www.leshatton.org/Documents/LOC2005.pdf.

[17] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS 18*, 2018.

[18] Y. Jiang, Q. Bao, S. Wang, X. Liu, and D. Wu. Reddroid: Android application redundancy customization based on static analysis. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 189–199, Oct 2018.

[19] Corey Kallenberg, John Butterworth, Xeno Kovah, and Sam Cornwell. Cve-2013-3582. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-3582.

[20] Corey Kallenberg, John Butterworth, Xeno Kovah, and Sam Cornwell. Defeating signed bios enforcement, Jan 2014.

[21] Colin Ian King. Firmware test suite, Feb 2020. https://wiki.ubuntu.com/FirmwareTestSuite.

[22] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. Attack surface metrics and automated compile-time OS kernel tailoring. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.

[23] Brad Linder. Intel plans to end legacy BIOS support by 2020, November 2017. https://liliputing.com/2017/11/intel-plans-end-legacy-bios-support-2020.html.

[24] John Loucaides and Yuriy Bulygin. Platform Security Assessment with CHIPSEC. *CanSecWest 2014*, March 2014.

[25] Steven C. MacConnell. *Code complete: a practical handbook of software construction*. Microsoft Press, 2004.

[26] Maksim Malyutin. Cve-2017-5689. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5689.

[27] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, USA, 2004.

[28] Ghassan Misherghi and Zhendong Su. HDD: hierarchical delta debugging. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*, Shanghai, China, 2006. ACM Press.

[29] Bruce Monroe, Rodrigo Rubia Branco, and Vincent Zimmer. Firmware is the new black – analyzing past 3 years of bios/uefi security vulnerabilities, Jul 2017.

[30] Karsten Nohl and Jakob Lell. Badusb - on accessories that turn evil, Jul 2014. https://srlabs.de/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf.

[31] Bazhaniuk Oleksandr, Bulygin Yuriy, Andrew Furtak, Mikhail Gorobets, John Loucaides, Alex Matrosov, and Mickey Shkatov. Attacking and Defending BIOS in 2015. *RECON 2015*, June 2015.

[32] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through piece-wise compilation and loading. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 869–886, Baltimore, MD, August 2018. USENIX Association.

[33] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick Mcdaniel. Cimplifier: automatically debloating containers. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, 2017.

[34] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming. *ACM Transactions on Information and System Security*, 15(1):1–34, Mar 2012.

[35] Palsamy Sakthikumar and Vincent J. Zimmer. White paper: A tour beyond bios implementing the acpi platform error interface with the unified extensible firmware interface. Technical report, Intel Corporation, January 2013. https://firmware.intel.com/sites/default/files/resources/A_Tour_beyond_BIOS_Implementing_APEI_with_UEFI_White_Paper.pdf.

[36] Nikolaj Schlej. Uefi tool. https://github.com/LongSoft/UEFITool.

[37] Nikolaj Schlej. Analyzing the source code of uefi for intel galileo by pvs-studio, May 2015.

[38] Nikolaj Schlej. Zero nights, Nov 2015.

[39] Spotify. spotify/luigi, Jun 2019. https://github.com/spotify/luigi.

[40] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 361–371, New York, NY, USA, 2018. ACM. http://doi.acm.org/10.1145/3180155.3180236.

[41] Tianocore. Pi boot phase, 2019. https://raw.githubusercontent.com/tianocore/tianocore.github.io/master/images/PI_Boot_Phases.JPG.

[42] Matthew Weeks. *Network Nightmare*. Aug 2011.

[43] Stefan Weil. Qemu user manual. https://qemu.weilnetz.de/doc/qemu-doc.html.

[44] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. http://dl.acm.org/citation.cfm?id=800078.802557.

[45] Corey Kallenberg Xeno Kovah. How Many Million BIOSes Would you Like to Infect?, 2015. http://legbacore.com/Research_files/HowManyMillionBIOSesWouldYouLikeToInfect_Whitepaper_v1.pdf.

[46] Andreas Zeller. Simplifying and Isolating Failure-Inducing Input. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 28(2):17, 2002.

Appendix: Pruning Results

Table 1: Modules removed

Motherboard	Pruning Mode	Original modules	Remaining modules	Reduction
SuperMicro A1SAi-2550F (V519)	Aggressive	244	90	63.11%
Tyan 5533V101	Aggressive	194	60	69.07%
HP DL380 Gen10	Aggressive	643	323	49.77%
SuperMicro A1SAi-2550F (V827)	Data Center	241	124	48.55%
SuperMicro A2SDi-12C-HLN4F	Data Center	313	194	38.02%
SuperMicro A2SDi-H-TP4F	Data Center	313	206	34.19%
SuperMicro X10SDV-8C-TLN4F	Data Center	316	286	9.49%

Table 2: Gadgets removed

Motherboard	Pruning Mode	Original	Pruned	Reduction
SuperMicro A1SAi-2550F (V519)	Aggressive	78389	28414	63.75%
Tyan 5533V101	Aggressive	73203	40212	45.07%
HP DL380 Gen10	Aggressive	369663	216831	41.34%
SuperMicro A1SAi-2550F (V827)	Data Center	77929	46680	40.10%
SuperMicro A2SDi-12C-HLN4F	Data Center	89736	64267	28.38%
SuperMicro A2SDi-H-TP4F	Data Center	90566	64177	29.14%
SuperMicro X10SDV-8C-TLN4F	Data Center	109680	96239	12.25%

Table 3: Gadgets removed (high quality)

Motherboard	Pruning Mode	Original	Pruned	Reduction
SuperMicro A1SAi-2550F (V519)	Aggressive	37846	14240	62.37%
Tyan 5533V101	Aggressive	38776	20317	47.60%
HP DL380 Gen10	Aggressive	183677	105116	42.77%
SuperMicro A1SAi-2550F (V827)	Data Center	37735	23055	38.90%
SuperMicro A2SDi-12C-HLN4F	Data Center	43593	31003	28.88%
SuperMicro A2SDi-H-TP4F	Data Center	44121	31024	29.68%
SuperMicro X10SDV-8C-TLN4F	Data Center	51534	45724	11.27%

Table 4: Byte surface area reduction

Motherboard	Pruning Mode	Byte SA (kb)	Remaining byte SA (kb)	Reduction
SuperMicro A1SAi-2550F (V519)	Aggressive	3013	903	70.91%
Tyan 5533V101	Aggressive	4520	1916	39.82%
HP DL380 Gen10	Aggressive	46102	27809	39.68%
SuperMicro A1SAi-2550F (V827)	Data Center	3000	2108	29.76%
SuperMicro A2SDi-12C-HLN4F	Data Center	3618	2680	25.91%
SuperMicro A2SDi-H-TP4F	Data Center	3645	2766	24.12%
SuperMicro X10SDV-8C-TLN4F	Data Center	4519	4209	6.87%

Table 5: Estimated defects removed

Motherboard	Pruning Mode	LoC (est.)	LoC Removed (est.)	Defects removed (est.)
SuperMicro A1SAi-2550F (V519)	Aggressive	215235	150755	2261
Tyan 5533V101	Aggressive	322870	186049	2791
HP DL380 Gen10	Aggressive	318571	55071	826
SuperMicro A1SAi-2550F (V827)	Data Center	214307	63736	956
SuperMicro A2SDi-12C-HLN4F	Data Center	258429	67000	1005
SuperMicro A2SDi-H-TP4F	Data Center	260357	62786	942
SuperMicro X10SDV-8C-TLN4F	Data Center	322786	22143	332

Table 6: Comparison of EFI images from different vendors

	ASRock	Asus	EVGA	Gigabyte	SuperMicro
Number of modules	962	362	443	461	386
ASRock	X	257/25%	108/11%	280/29%	198/20%
Asus		X	135/30%	256/55%	183/47%
EVGA			X	106/23%	77/17%
Gigabyte				X	245/53%

Table 7: Comparison of 7 random firmware images from ASRock

ASRock	AB350M	B365M	B450	Fatal1ty_Z370	H110M-HDV	IMB-390-L	Z390
Number of modules	466	883	641	942	605	328	941
AB350M_Pro4_DASH	X	212/24%	452/70%	210/22%	200/33%	196/42%	208/22%
B365M_Pro4		X	394/44%	860/91%	540/61%	269/30%	856/90%
B450_Steel_Legend			X	392/41%	302/47%	190/29%	392/41%
Fatal1ty_Z370				X	557/59%	267/28%	850/90%
H110M-HDV_R3.0					X	294/48%	530/56%
IMB-390-L						X	270/28%

Table 8: Comparison between different patches of ASRock IMB186 motherboard

ASRock	V1.1	V1.4	V1.5	V1.6	V1.7	V1.8	V1.9	V2.1	V2.3
Number of modules	257	257	257	289	268	268	268	299	299
V1.1	X	100%	100%	257/88%	257/95%	257/95%	257/95%	257/85%	257/85%
V1.4		X	100%	257/88%	257/95%	257/95%	257/95%	257/85%	257/85%
V1.5			X	257/88%	257/95%	257/95%	257/95%	257/85%	257/85%
V1.6				X	258/89%	258/89%	258/89%	289/96%	289/96%
V1.7					X	100%	100%	268/89%	268/89%
V1.8						X	100%	268/89%	268/89%
V1.9							X	268/89%	268/89%
V2.1								X	100%