Electronic Design

SWASTIK GHOSH, Firmware Engineer – UEFI, Lenovo

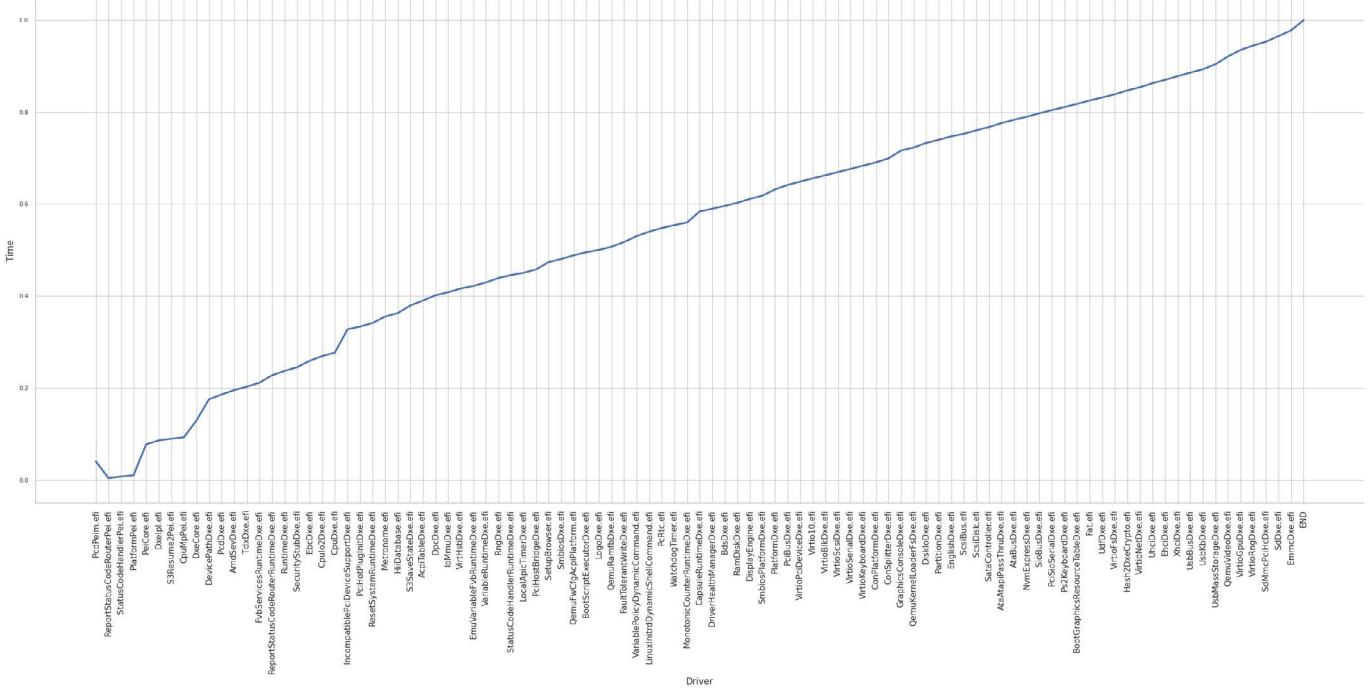
Same Firmware, Different Behavior: Understanding Configuration Effects in EDK2

Firmware configuration is more than just enabling features — it can quietly reshape execution or fundamentally alter system behavior, as revealed through an analysis of EDK2 firmware.

Firmware configuration is often assumed to be a static, low-risk aspect of system development. Developers enable or disable features through build-time options. They expect predictable and incremental changes in behavior. However, these choices can have subtler and sometimes unexpected effects on how a system executes during early boot stages.

On Universal Extensible Firmware Interface (UEFI)-based platforms, particularly those built on EDK2, configuration parameters, particularly those stored in the Platform Configuration Database (PCDs), control a wide range of functionality. This includes bringing up the network stack, implementing secure boot mechanisms, supporting trusted platform modules, and enabling system-management features. TianoCore is the open-source implementation of EDK2.

While these options are typically evaluated in terms of their functional impact, their influence on runtime execution is far less understood. Traditional debugging strategies rely on linear logs or total boot-time measurements, which



1. Plot depicting the time series representation of a single boot iteration. (Credit: Lenovo)

commonly don't capture how execution behavior evolves across several platform configurations.

That visibility gap is important when dealing with performance tuning, platform bring-up, or intermittent boot issues. Two systems built from the same codebase and differing only in configuration may show different execution characteristics, even without a clear change in boot time. Some configurations add disproportionate overhead, which can dramatically alter system operation. These effects often aren't obvious without deeper analysis.

In this article, we take an organized approach to understanding how firmware configuration affects runtime behavior. Using an instrumented EDK2 build, we will analyze multiple boot runs across configurations. This lets us go beyond basic metrics and see execution patterns in greater detail. The results show that firmware configuration can cause subtle changes in execution sequence, or even a complete shift in how a system behaves during boot.

Experimental Setup: Measuring Firmware Execution

To analyze how firmware configuration impacts runtime behavior, experiments were performed using the EDK2 OVMF (Open Virtual Machine Firmware, an EDK2-based UEFI implementation for virtual machines) package on a QEMU-based environment. This setup delivers a controlled platform for observing firmware execution over various configurations.

To capture execution behavior, the EDK2 codebase was instrumented during the PEI and DXE phases. At each stage, the system records the firmware component being loaded. It also saves a timestamp showing when execution moves to the next component. This allows us to reconstruct a detailed timeline of firmware execution, including both the order and time spent between modules.

Each configuration was executed across multiple boot runs to capture subtle pattern shifts and to create a unique execution profile. By aggregating data from multiple runs, we could identify reliable execution patterns and pinpoint subtle differences caused by configuration changes.

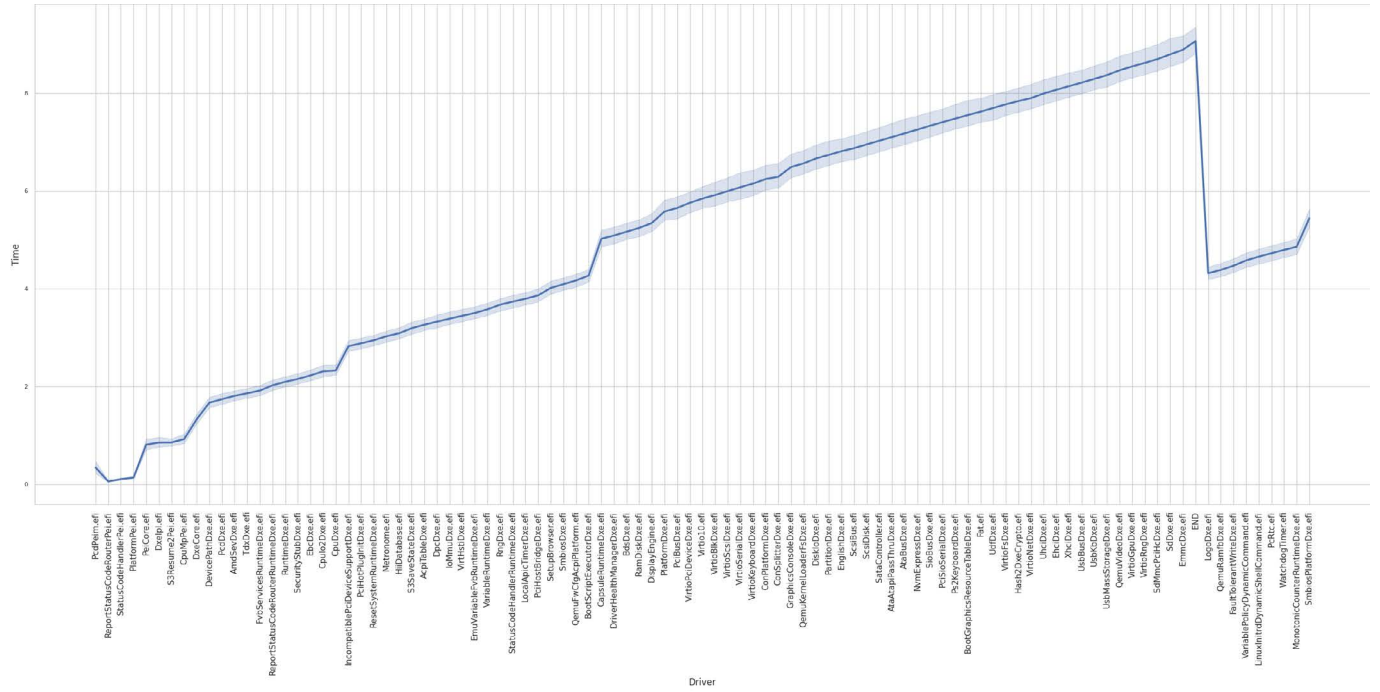
The following EDK2 configurations were evaluated:

- Network TLS enabled
- Secure Boot enabled (with and without platform keys)
- System Management Mode (SMM) enabled
- Trusted Computing Group (TCG), an industry group that sets standards for secure computing, is stack-enabled with SMM.

This mix of configurations lets us compare feature enablement settings with security-oriented settings. It offers insight into how both types affect runtime behavior.

Observing Execution: From Logs to Behavior

After obtaining the timestamped log, the next step was to reshape it to make execution behavior clearer. Each log entry includes the name of the loaded EFI component and the time it was loaded. This information allowed us to rebuild a time-ordered sequence of the firmware's execution.



2. Multiple boot iterations aggregated into a single graph. (Credit: Lenovo)

By plotting these events, we constructed a time series representing the boot *(Fig. 1)*. Each point shows the execution of a specific EFI component. This gives a continuous view of boot progress through the two crucial stages: Pre-EFI (PEI) and Driver Execution Environment (DXE).

Since EFI firmware executes sequentially, the time between graph points shows the cost of running each EFI component. When visualized, this forms a unique execution profile. Areas of dense activity and delay become apparent in the graph.

To account for variation across boot iterations, we aggregated execution data from multiple runs of the same configuration into a single visualization *(Fig. 2)*. This shows the expected pattern for a given configuration. It accounts for minor fluctuations but preserves consistent structural behavior.

When comparing aggregated profiles across configurations, clear differences emerge. Many configurations produce similar execution profiles. However, localized variations appear in specific regions. These shifts reflect changes in process flow, often driven by new components or alternative initialization paths.

By correlating those deviations with the logs, we can find the specific EFI modules that cause the changes. This approach turns raw logs into a systematic way to track how firmware behavior changes across configurations. It serves as the basis for deeper analysis in the following sections.

Subtle Changes: When Behavior Shifts Without Time Impact

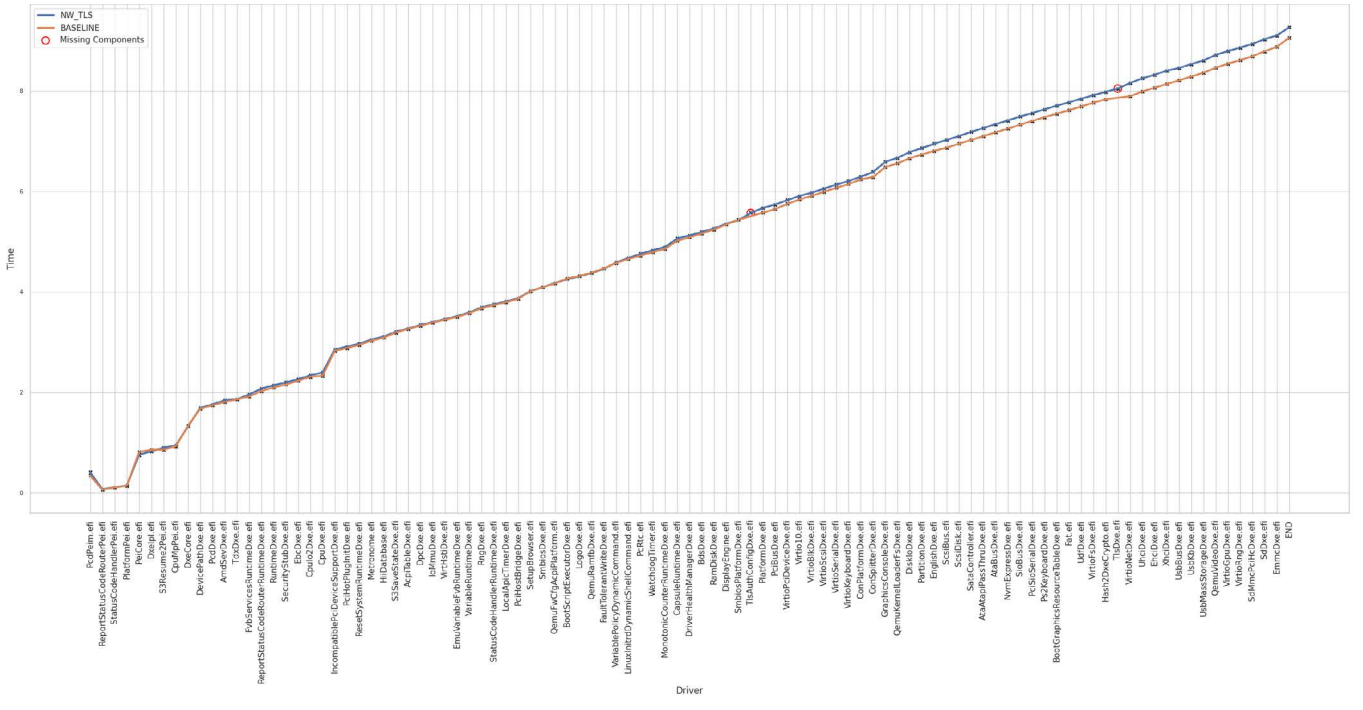
To see how firmware configuration affects execution without changing total runtime, we examined cases with functional changes but little impact on boot time. Here, we look at two examples: enabling network TLS and enabling Secure Boot. We also compare Secure Boot with and without platform keys.

A baseline configuration was set up with all optional features off. This serves as a control for comparing changes in later configurations. Enabling network TLS changed the execution profile in certain spots, compared to the baseline *(Fig. 3)*.

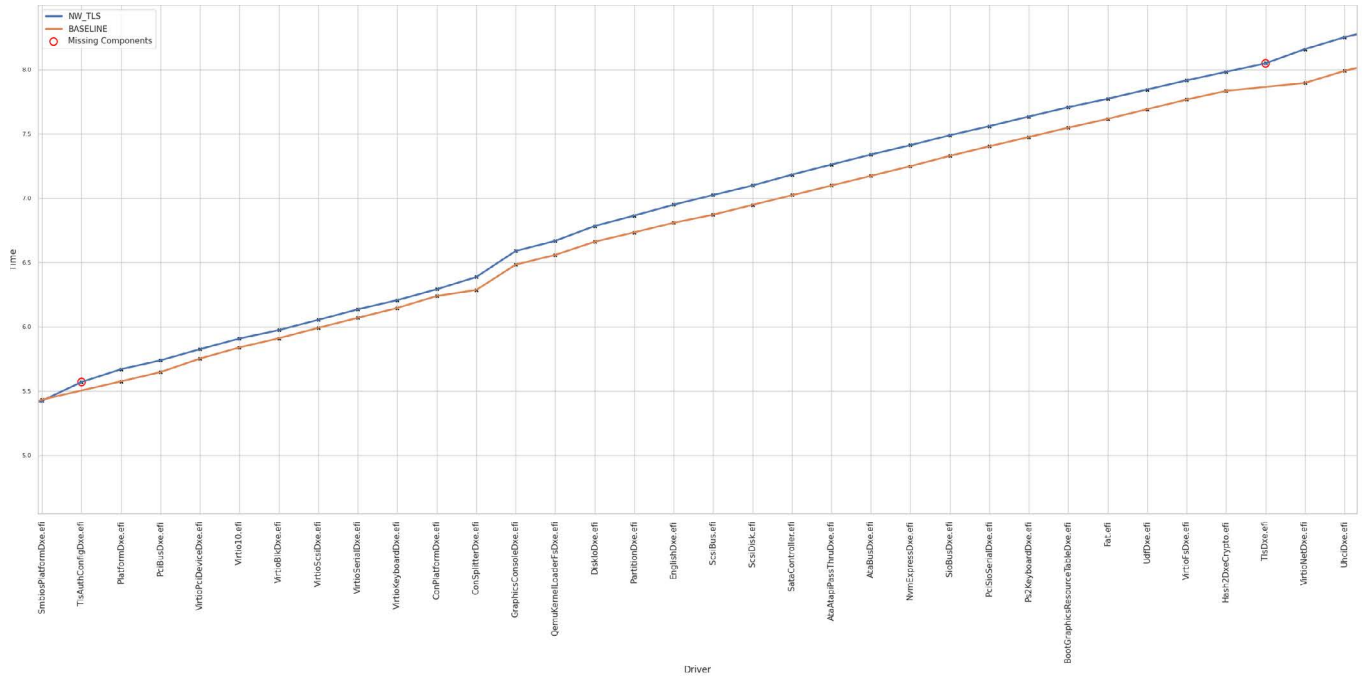
A deeper look at the logs showed extra EFI components. These included TlsAuthConfigDxe.efi and TlsDxe.efi, highlighted in *Figure 3* by red circles. These components adjusted the execution structure by extending or shifting some parts of the timeline. Still, there was no overall increase in boot time.

A similar pattern was observed when Secure Boot was enabled *(Fig. 4)*. Additional components, such as SecureBootConfigDxe.efi and IgvmSecureBootDxe.efi, were introduced into the execution path, modifying the sequence and distribution of work during the DXE phase. Yet, as with the network TLS configuration, these changes didn't result in a measurable increase in total boot time.

To check if runtime state matters, Secure Boot was tried



3. Comparison of the baseline boot with a boot iteration having Network TLS enabled. (Credit: Lenovo)

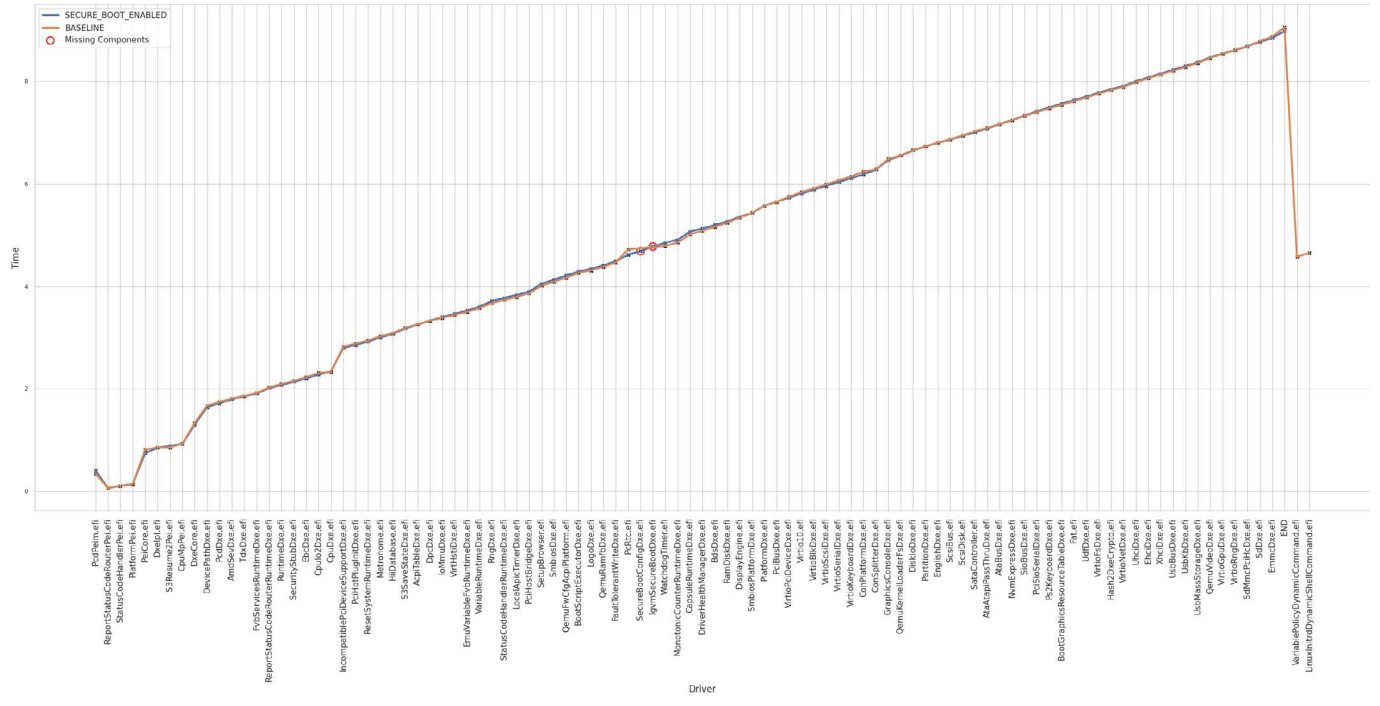


4. Zoomed comparison of network TLS and baseline boot, indicating components missing in the baseline. (Credit: Lenovo)

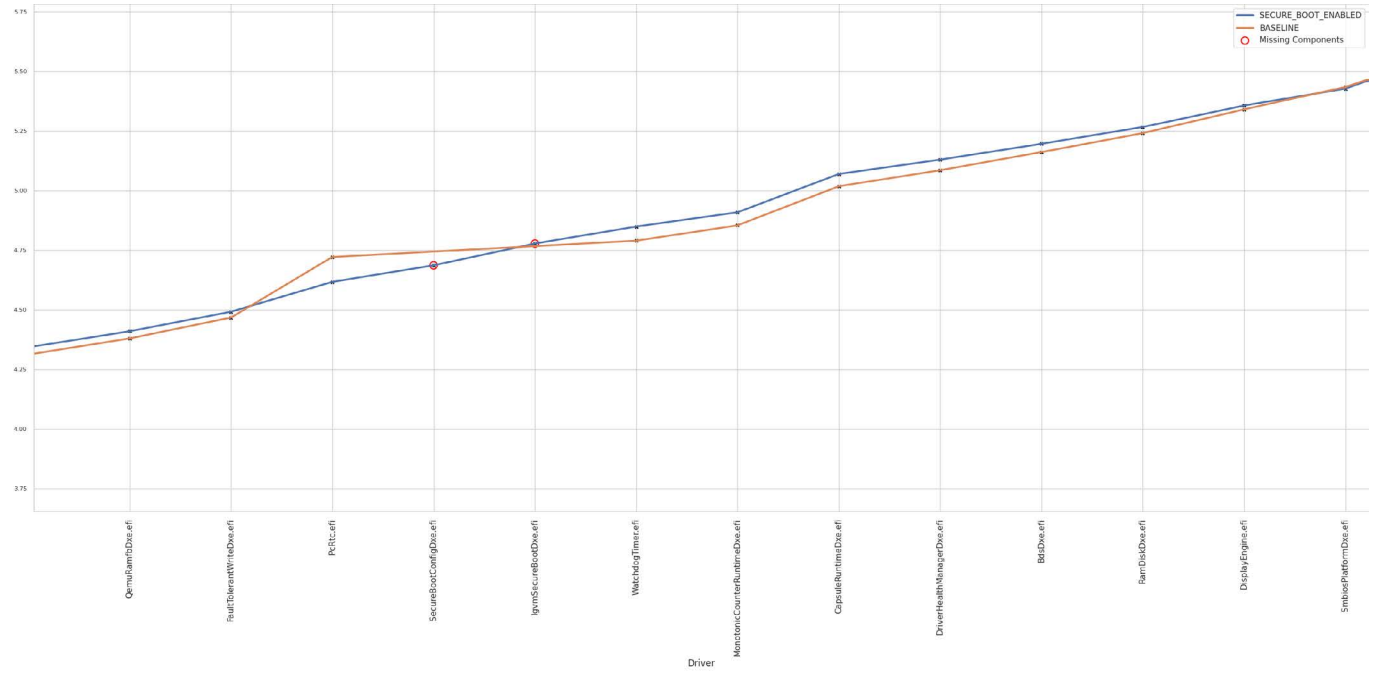
with platform keys installed *(Figs. 5 and 6)*. This switch moves the system from setup to enforcement mode, enabling signature verification. Despite this, the execution profile stayed broadly similar to the non-keyed setup. There were no major differences in runtime or execution structure.

These results highlight a key feature of firmware behavior. Configuration changes can reshape program flow without affecting overall performance measures *(Fig. 7)*. New components can enter and routes can shift, but the system adjusts within its existing time bounds.

While these setups don't immediately expose performance bottlenecks, they afford valuable insight into how execution paths evolve. In particular, they reveal parts of the boot process that are sensitive to configuration changes, emphasizing



5. Comparison of a Secure Boot run with the baseline showing changes in the shape with the introduction of new Secure Boot components. (Credit: Lenovo)

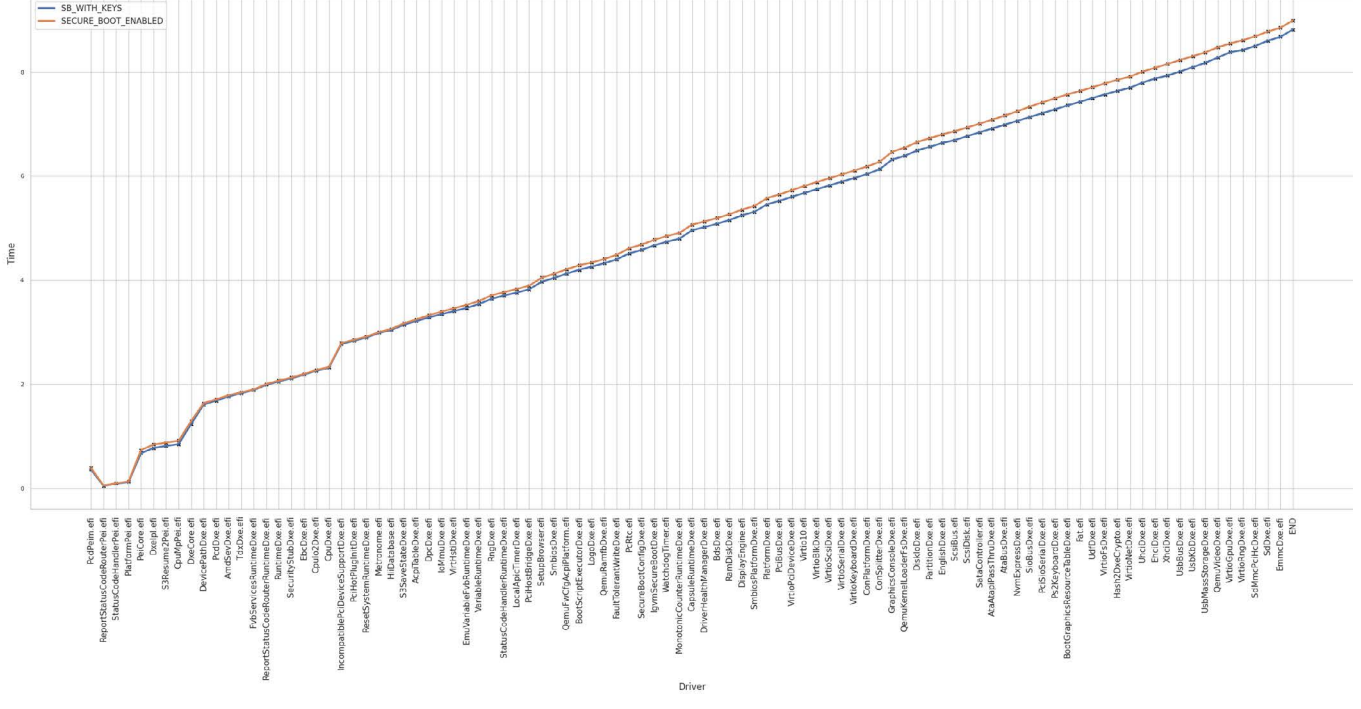


areas that may become more critical when more complex or resource-intensive features are introduced, as explored in the following sections.

Configuration vs. Runtime State

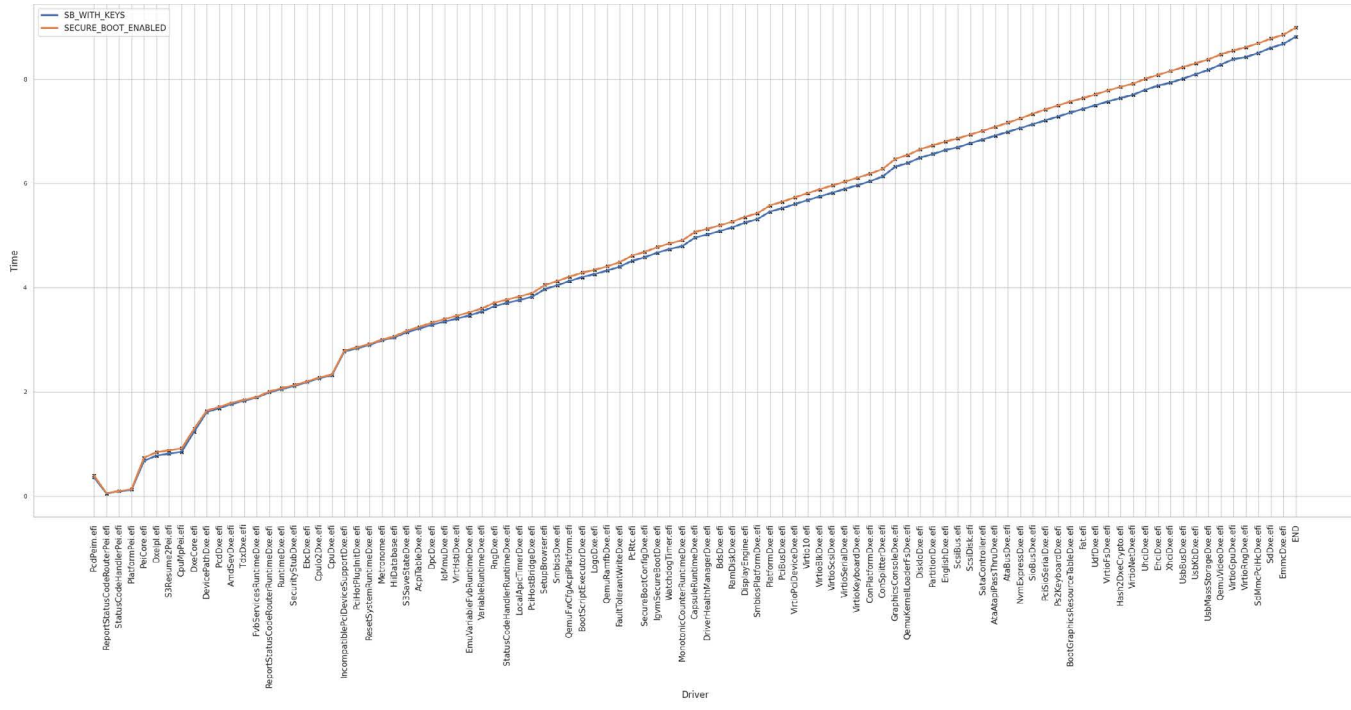
Upon examining Secure Boot, it's clear that the system's capabilities are determined not only by configuration flags, but also by the platform's state. This was determined by evaluating Secure Boot in two scenarios: having the feature keys provisioned. Although both configurations were built from the same codebase and enabled the same feature set, they represent two fundamentally different operating states.

In the first case, without platform keys, the system operates in the state commonly referred to as setup mode, where the Secure Boot components are present and initialized, but signature verification isn't enforced. The execution profile reflects the inclusion of Secure Boot modules, but without bringing significant changes to runtime behavior.



enabled without the platform keys installed, and with the

On the other side, once the platform keys are installed, the

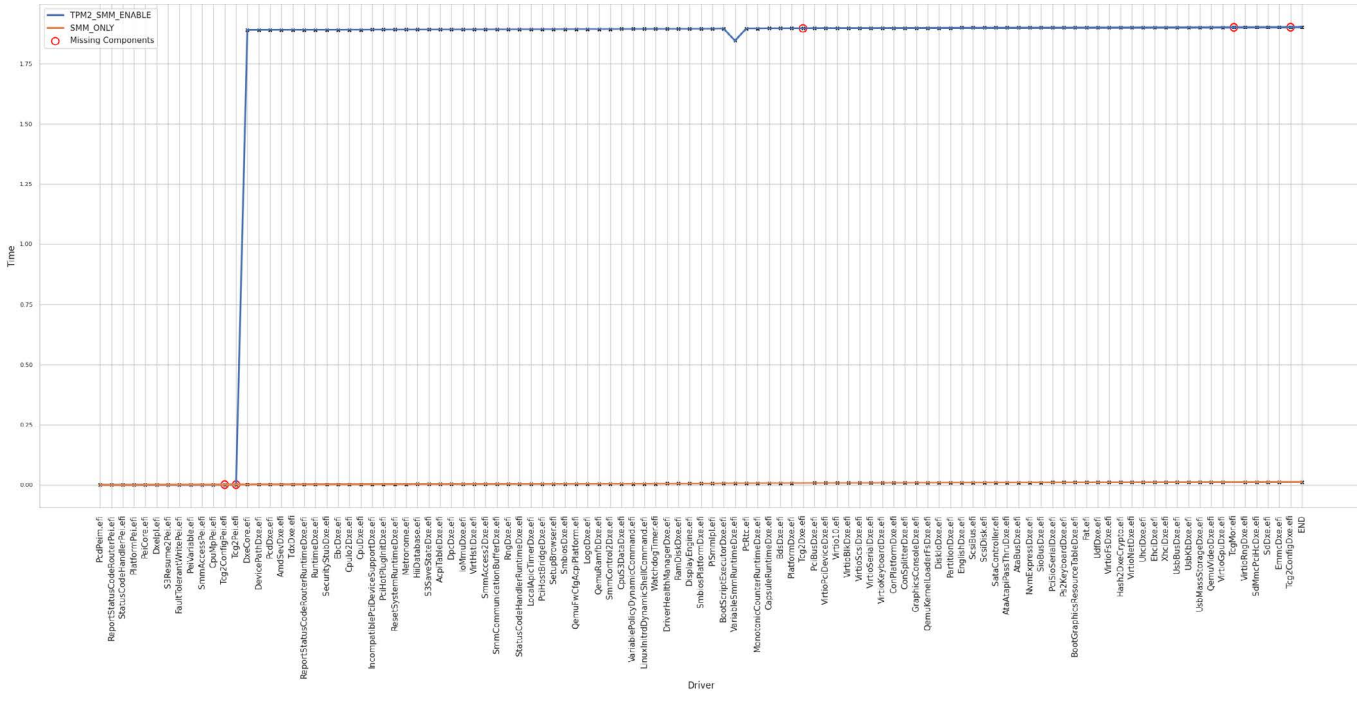


6. Zoomed comparison of a Secure Boot run with the baseline, indicating the components missing from the baseline. (Credit: Lenovo)

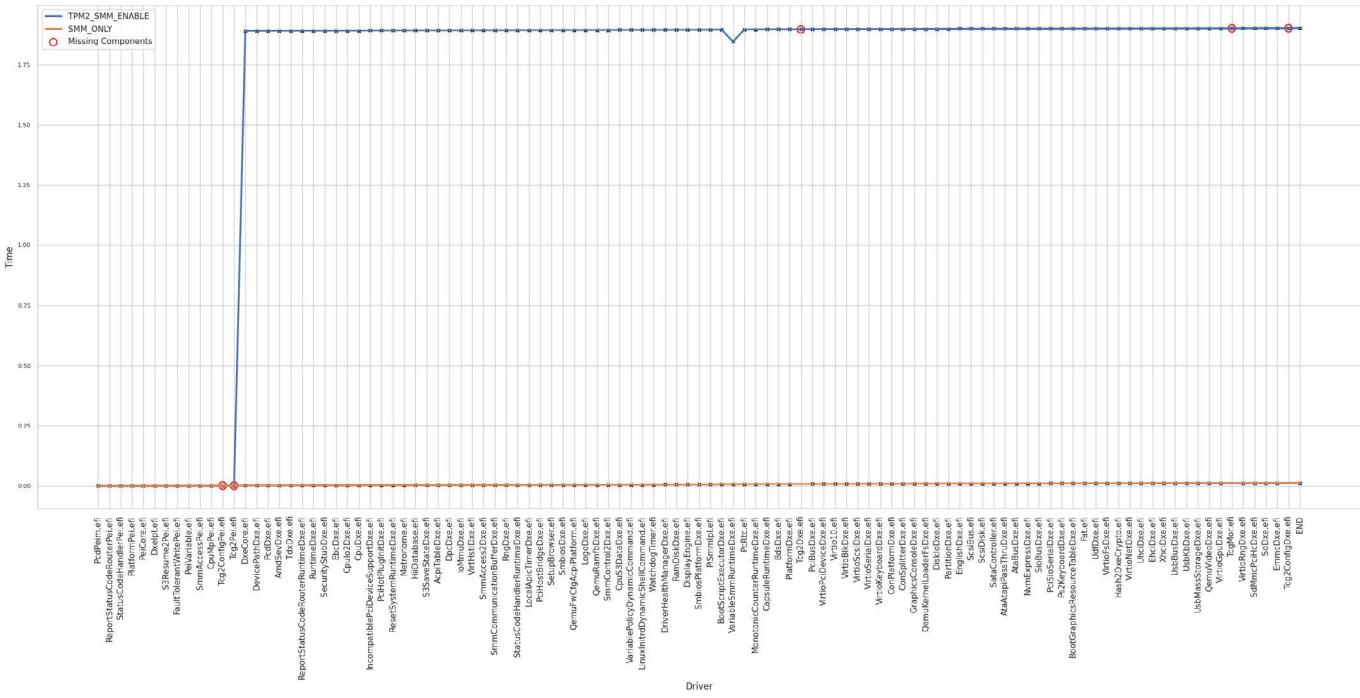
system moves to enforcement mode, where signature verification becomes active. This adds extra authentication steps during execution, particularly when loading and verifying firmware components. However, when comparing the execution profiles of the keyed and non-keyed configurations, the overall structure and total boot time remain largely consistent.

It becomes apparent that an important characteristic of firmware behavior involves the runtime state, which can alter how features are applied without introducing any significant performance cost. Even with additional security mechanisms in place, their impact may be distributed across the execution path, with little effect on aggregate metrics.

More broadly, this demonstrates that configuration alone



7. Comparison of two Secure Boot runs with and without platform keys installed. (Credit: Lenovo)



8. Significant shift in the boot trajectory with the TCG stack enabled (blue), as compared to the SMM-only run (orange). (Credit: Lenovo)

is insufficient to fully describe system operation. An identical firmware build can exhibit different execution characteristics depending on how it's provisioned and initialized at runtime. Thus, it emphasizes the need to consider configuration and state when analyzing system operation, especially when working with security-sensitive features.

When Configuration Becomes Dominant

While many firmware configurations result in minor changes in execution behavior, some can radically alter how the system behaves during boot. In these cases, the configuration doesn't simply reshape execution; rather, it becomes the dominant behavior at runtime.

To explore this, System Management Mode (SMM) and the Trusted Computing Group (TCG) stack were evaluated both independently and in combination. When SMM was executed standalone, the system showed behavior similar to the baseline configuration, with total boot time remaining on the order of around a second. The execution profile showed only minor structural differences, indicating that SMM alone doesn't cause any significant overhead in this environment.

However, when the TCG stack was enabled alongside SMM, the behavior changed dramatically. The overall execution time of the PEI + DXE phase increased from approximately a second to a couple of minutes, representing a shift that's orders of magnitude larger than any previously observed configuration effect *(Fig. 8)*.

Unlike the other cases, this isn't simply a change in the overall boot structure — it revealed that the slowdown occurred at a particular stage during PEI. In particular, when the TCG PEI component is loaded with SMM enabled, it silently enables code sections responsible for secure variable measurement and initialization, exhibiting a strong correlation between the TPM-based measurement and SMM execution.

A further investigation of the logs indicated an extended execution within these components, pointing to either intensive cryptographic operations, repeated variable measurements, or blocking interactions with the underlying platform emulation. While the precise cause may depend on the specifics of the environment, the main observation is consistent: The system enters a fundamentally different execution regime when these features are combined.

This contrast between SMM alone and SMM + TCG stack highlights an important characteristic of firmware configuration. Interactions between features can have nonlinear effects. Individual features may appear lightweight in isolation. However, when combined, they can bring significant overhead due to compounded responsibilities, such as measurement, validation, and secure state management.

From the perspective of a firmware developer, this stresses

the importance of evaluating configurations not only in isolation, but also in combination. Performance characteristics can't always be inferred from individual features alone, particularly when security mechanisms are involved. Instead, it's necessary to study how these features engage within the wider process flow.

Two Modes of Firmware Behavior

Observations throughout different configurations reveal a more extensive pattern in how firmware behaves during execution. Instead of responding uniformly to configuration changes, firmware operates in separate modes depending on the nature and interaction of enabled features.

The first mode can be thought of as a structural change. In this mode, configuration options add extra components or alter the execution path without affecting runtime. This behavior was observed in configurations such as network TLS and Secure Boot, where new execution paths were modified, yet total boot time remained largely unchanged. The system modifies through redistributing work across the existing execution timeline, preserving overall performance while altering its internal structure.

The second mode represents a shift in the execution regime. In this case, configuration changes don't merely modify execution — they dominate it. It was evident when the TCG stack was enabled alongside SMM, resulting in a dramatic increase in boot time. In this case, the system was no longer balancing additional work within its processing sequence. Rather, specific components, especially those related to secure measurement and validation, became the primary drivers of runtime behavior.

It's evident from studying those two modes that not all configuration changes are equal. Some operate within the existing performance envelope, subtly reshaping execution without increasing total runtime. Others introduce fundamentally different execution characteristics, where the cost of additional functionality outweighs the system's ability to absorb it.

Engineers commonly rely on high-level metrics, such as total boot time, to evaluate the impact of configuration changes. However, the structural change mode shows that substantial changes in execution behavior can occur without affecting these aggregate metrics. Conversely, in the regime shift mode, performance degradation is obvious but may not immediately reveal its underlying cause without deeper analysis.

Engineering Implications

The observations presented in this study show an important gap between how firmware evaluation is typically evaluated and how it actually affects system operation. In practice, configuration changes are often assessed using classic standards such as total boot time or basic functional validation. While these metrics are useful, they may obscure more subtle but meaningful changes in execution behavior.

A key takeaway is that execution structure matters as much as execution time, as shown in the structural change mode. Configurations such as network TLS and Secure Boot bring in new components and alter the operation sequence without substantially impacting overall runtime. However, it's these components, which are otherwise hidden, that could considerably alter overall boot time, while the initial impression might suggest that a configuration change has a negligible impact.

Another major implication relates to feature interaction, as seen with the combination of the TCG stack and SMM enabled, where nonlinear effects can arise. Features that appear lightweight in isolation may cause substantial overhead when their responsibilities overlap, notably in areas like security, measurement, and state management. This makes it hard to predict system behavior solely from individual features.

For a firmware developer, it signals the need to evaluate conditions not just independently, but also in realistic combinations. Testing strategies should account for these interplays, especially within systems where security features are layered on top of existing functionality.

These conclusions show the importance of structured observability in firmware environments, particularly in EFI firmware. Traditional debugging approaches that rely on logs and intuition are often not sufficient to detect the fine details of execution behavior. Therefore, by instrumenting firmware and analyzing execution across multiple runs, it becomes possible to identify reliable patterns, detect structural changes, and reason about system operation in a more systematic way.

Rethinking Firmware Configuration

Firmware configuration is often treated as a static, predictable aspect of system design, in which functionality can be enabled or disabled without radically changing how the system behaves. However, the findings of this study would challenge this assumption.

Across multiple configurations, it's clear that firmware doesn't respond uniformly to configuration changes. Some configurations introduce minor changes in execution, adding or reordering components without changing overall runtime. Others, notably those involving security features and their interactions, can drastically alter execution behavior, bringing about significant changes in performance and system dynamics.

Configuration isn't simply a build-time concern, but a runtime factor that determines behavior. The same firmware codebase, when executed under different configurations and

states, can exhibit distinct execution characteristics, even when high-level metrics similar to overall boot time remain unchanged.

By instrumenting and analyzing firmware execution across multiple runs, it becomes possible to move beyond cursory observations and develop a more structured understanding of how systems behave during early boot. This would still remain hidden in traditional logs, allowing engineers to spot subtle changes, identify anomalies, and reason about complex feature associations.

As firmware systems become more complex, particularly in security and platform integrity, such a level of insight becomes increasingly important. Understanding not just what a system does, but also how it behaves under different configurations. This would be a keystone for building reliable, predictable, and high-performance platforms.

Ultimately, this work implies a shift in viewpoint: Firmware configuration should not be viewed as a static set of options, but rather as a mechanism that defines how the system operates among different configurations. Recognizing this allows for more effective debugging, better performance analysis, and a fuller understanding of the systems we build.

Swastik Ghosh is a firmware engineer with experience in low-level software development, Real Time Operating Systems as well as in UEFI development. He works on analyzing and optimizing firmware execution across different configurations, with an emphasis on performance, reliability and system level interactions.