Architecting for Secure, Safe and Agile Software Defined Vehicles

Learnings from the Personal Computer industry

Sean Selitrennikoff, Microsoft, Redmond, WA, USA; Mario Ortegón-Cabrera, Microsoft, Aachen

Abstract

Traditionally, automotive companies design vehicles as an independent entity which have mostly unchanging capabilities and experiences at start-of-production. However, the automotive customer has become more sophisticated and demands the ability to continuously evolve the capabilities and experience of the vehicle. To meet this challenge, automotive software and hardware must be architected from the ground-up to be secure, reliable, safe and extensible.

The automotive architecture must be coupled with modern development processes that enable an agile software development and deployment, reduce complexity, and allow the integration of third-party software assets without compromising functional safety, security, or reliability of other assets.

These challenges are very similar to the challenges the PC industry faced as it matured. We will explore the parallels and evaluate from a perspective of security, reliability, and agility. We will wrap up by taking what has been learned and apply it to vehicle hardware and software design, showing one example of a hardware-plus-software architecture that meets the challenges of today.

Introduction

In today’s world, creating a car is a long-drawn out process. The definition of the future vehicle functions takes place years before the chassis hits the production line. The vehicle electric/electronic architecture is created specifically for a product family and it remains stable for several years, to be replaced by a full redesign at the end of the product lifecycle. As software became more prevalent in cars, automotive software engineering followed other
automotive engineering disciplines and started with a sequential, step-by-step waterfall model with Gantt Diagrams.

This approach was very successful for many decades, but was disrupted when a new approach was taken, the software defined vehicle. By adopting modern software and hardware, the software defined vehicle revolutionized the industry, raising the expectations of consumers and the bar for all car manufacturers. This approach has proven to be very successful, as Tesla, for example, reached a market capitalization of 615 billion dollars in November 2020, surpassing the next 7 automakers combined (1).

In this manuscript we will discuss important parallels between the challenges faced by the personal computing industry and the automotive industry. Thankfully the PC industry has had decades to figure out great ways to address these challenges. This enables the automotive industry to accelerate its maturation and guide the development of secure, safe, and agile software defined vehicles. We will start by looking at important inflection points in the history of the PC industry, the learned lessons and impact on the hardware and software architectures of the PC. We will finish by describing an automotive hardware and software architecture, as well as development process.

**Evolution of hardware and software in the Personal Computer industry**

To look into the future, it is useful to analyze the past and compare with adjacent industries. The PC industry faced many challenges as it grew from an isolated computer in a home, to a networked and internet connected, multi-user, computational marvel. There are several points in time where new technology and new availability created a large, new, set of challenges – and we shall examine each of these to see what learnings can accelerate the automatic industry’s hardware and software maturation.
Our first stop in looking back at the PC industry is the **1980s**, when the first personal computers hit the market with a common operating system and open hardware architecture. This created the first general computing platform available to the general population. An important aspect of this general computing platform was that it was extensible in both hardware and software. There was one drawback, though. This general computing platform was expensive. Some companies saw this as an opportunity to build special purpose computing devices for specific use cases, such as word-processing, at a lower price point. However, the users soon discovered that the flexibility of the PC enabled them to buy new software quickly and cheaply. This allowed them to “future-proof” their investment and do things they could not do before. Special purpose devices were mostly a dead-end and general computing won out.
These first computers had very unreliable software experiences. An application had full control over the processor, memory, and storage. There was no concept of security, other than restricting physical access to the machine. Software agility was very low because few companies made software for the PCs. Software was only available at a few specialized marketplaces. Hardware was difficult to configure and required specialized knowledge: Technicians and advanced users had a wide array of dip switches and jumpers to set the configuration. Often hardware would conflict with other hardware and make the computer unusable.

Moving into the early 90s, PCs were predominantly using various flavors of Windows 3 and there was an explosion of available applications. Many companies were building ever-impressive software for Windows, with many stores and channels made available to consumers for purchasing software. In addition, there was tons of new hardware expansion cards also available. Overall, the agility of the platform was greatly improved from the 1980s.
To increase reliability, Windows did what it could to isolate applications from each other. Examples include virtual memory and CPU scheduling to provide physical access isolation, while abstractions like the Windows APIs and the Component Object Model (COM) provided software abstraction isolation. Although software reliability and agility were improved, there were still fundamental issues that negatively impacted reliability. One example is library dependency problems: Multiple applications relying on a single Dynamic Link Library (DLL) would break if the library was updated with a new version that was not backwards compatible.

On the security front, Windows adopted the concept of users with passwords. But Windows was still primarily dependent on physical access security.

Unfortunately, hardware reliability and security were essentially unchanged from the 1980s.

The mid 90s saw a major shift with the introduction of the Internet to consumers. In the context of this manuscript, the Internet enabled software distribution, direct and crowd sourcing problem solving and cooperative development. However, it also opened multiple new security attack vectors. Internet access forced companies like Microsoft to take security seriously, sparking the trustworthy computing pivot in development processes, tools, and verification (2). So, in some way one can say that PCs got less secure as they were first connected to the internet: phishing, bait-clicking, vulnerability exploits, etc.

PCs moved away from the MS-DOS/Windows operating system to the Windows NT-based operating system which was designed from the ground-up to provide strong abstraction and isolation. The operating system considered applications to be untrusted by default. Abstraction and isolation in the operating system as a first-class concept greatly increased the reliability of PCs, as well as increasing security and agility.

The mid 90s saw advancement on the hardware side, finally. Hardware Plug-and-Play standardized the way that devices advertised capabilities and configuration options. In addition, hardware become more secure with the introduction of Trusted Platform Modules (TPMs) and network security.
The 2000s saw the rise of cloud computing, which had a significant impact on PC software and hardware architecture. Cloud computing scaled out the reliability and security problems that individual PCs were having to a ‘cloud-scale’. This resulted in major efforts to develop new technologies to address these problems. We saw the start of containerization, clustering, secure-by-default and more. All of these concepts were integrated with and into PCs, blurring the line between the PC itself and the cloud.

This also happened with hardware. Scaling out hardware agility, replacement, in a secure manner for clouds, trickled down to PCs, enabling PCs to consider new hardware untrusted by default for the first time.

**Analogies to Vehicle Electronic Architecture**

The evolution of the PC industry has strong analogies to the evolution of the vehicle electronics architecture. Due to price considerations, vehicle hardware is usually designed as special-purpose computing devices with fixed functionality. As the complexity of the vehicles increase, the communication between electronic control units becomes fragile. A change in the functionality from one ECU could break the function of other ECUs in hard to predict ways. The transition to connected vehicles exposed the passengers to potential vulnerabilities to privacy and safety through attacks to the telematics and infotainment system (3).

To increase agility, vehicle electronics are now transitioning to general-purpose computing units with software that can be updated over-the-air and that require integration of software components from different teams. These modern computers benefit from many of the technology advances in hardware and software inherited from the personal computer industry.

**Windows NT Operating System Architecture**

In this section we will briefly discuss the Windows NT Operation System architecture, at a very high level. The goal is to give a basic understanding into how a modern operating system implements and enforces _untrusted by default_ with abstraction and isolation.

The challenges faced by the automotive industry are like the challenges faced when designing a modern operating system. An operating system must not have compromises in security. It must maintain a clear separation between privileged and non-privileged operations. It also must be flexible and provide well-known and documented application programming interfaces.
It must make it simple to develop and support third-party applications that do not compromise
the operation of the system, even in the case of erroneous or malicious behavior.

Let us look at a simplified block diagram of the architecture for Windows NT based operating
systems.

![Fig. 3: Windows NT Architecture](https://doi.org/10.51202/9783181023846-47)

Everything in kernel mode is a trusted zone. Components within the kernel provide abstractions
and capabilities to the system resources that they manage. The kernel provides a variety of
services including things like memory, disk, user, and process management. The development
process is rigorous, and it goes through a strong validation process. Programming errors can
have dire consequences to security, stability, and privacy and must be avoided at all costs.

The Hardware Abstraction Layer (HAL) is part of the kernel. It allows different hardware
configurations underneath the kernel without requiring modifications to the kernel. The HAL
exposes the hardware as an API – usually there is no direct access to the hardware.
Applications run in user mode, which is considered untrusted. They have access to a limited
set of interfaces, limited access to system data and for the most part have no direct access to
hardware. The kernel must ensure that all user processes are fully isolated and guarantee that if an application crashes, it will not bring other applications or the system down. It is thru isolation that this is achieved, using concepts like virtual address space, where each process has its own memory space, and unable to access the memory of another processes. Applications must communicate to each other through well-established channels. It is noteworthy that even processes bundled with operating system are in user space and subject to the same application model.

The separation between kernel and user space simplifies the development processes. 3rd party developers can create software for the user space in an agile way and update it frequently.

Automotive Architecture

In this section we will describe how to use these concepts when creating an architecture for the software defined vehicles.

Just like a PC, there are many different types of applications within a vehicle. Some are critical to safety, others ensure physical and software security, others are non-critical (like radio controls). Some of these applications will necessarily do a lot of processing. For example, the advanced sensors and cameras required for autonomous and assisted driving applications can easily generate several terabytes of data per day. Complex operations will be executed against this data, in vehicle, for offline and limited bandwidth reasons.

Many applications provide user facing services and experiences that are constantly updated. These applications must be loosely coupled to the vehicle sensors and actuators and have no impact on functional-safety relevant vehicle functions. Like PCs, a stable abstraction of the vehicle can address reliability, security and safety concerns. This allows a separation between the Automotive SPICE development process from the agile approach.

Rather than trying to present a complete architecture in one picture, we will present each layer one at a time, showing how they build on each other, starting from the bottom of the stack - the hardware of the vehicle.

Within the vehicle there are a very large number of sensors, actuators, microcontrollers and so forth. The microcontrollers are vital to the health and safety aspects of the vehicle, executing functions in real time.
The experience of the PC industry has shown that a standardized hardware Plug-and-Play model greatly increases the security, reliability, and agility of hardware systems. Therefore, we’d like vehicles to have a similar concept. One example is the collaboration between the MCVP team, our automotive partners, and the Azure Digital Twins team. This collaboration is to create a standardized Plug-and-Play language for automotive via the Digital Twin Description Language.

The next step in our automotive architecture is taking this hardware configuration and capability information and creating a standardized abstraction of the vehicle, much like the HAL. A Vehicle Abstraction Layer (VAL) models the sensors and actuators of the vehicle and provides a simplified digital representation that is common across all vehicle models. All communication to the vehicle is mediated by the VAL, ensuring that any request meets all functional safety preconditions. Workloads cannot invoke vehicle functions directly.

Analog to an operating system, the trusted zone has the responsibility to guarantee the integrity of the system and provide a foundation for the execution of custom workloads. It must provide the “heavy lifting” of managing and monitoring the untrusted zone. Two quick examples of services provided by the trusted zone. First, security services manage workload access to anything, including data, vehicle information, and communication with other workloads. The security host also manages a Trusted Platform Module (TPM) to store required certificates. Second, Over-The-Air update and configuration management of any software in the vehicle.

https://doi.org/10.51202/9783181023846-47
Das Erstellen und Weitergeben von Kopien dieses PDFs ist nicht zulässig.
With the foundation pieces in place, there is now an untrusted zone for hosting untrusted applications. **Workloads must run in the Untrusted Zone.** This zone provides isolation that guarantees workloads have a separate memory space, do not access the same file system, and have separate libraries and dependencies. The system must also be able to control resource consumption and monitor the operation. The more isolation between the untrusted applications, the greater the security, reliability, and agility of the overall system. One way of maximizing isolation is to containerize each application. Therefore, a containerized space within the untrusted zone is highly desired, such as Moby, Docker, or Podman.

This architecture is great. However, we can do better. Let’s dive into these microcontrollers for a second to understand how. Today microcontrollers host functions that need real-time response times. This includes not only health and safety functions, but also non-health and safety functions that need response times perceptible by humans (like body control functions). Unfortunately, by putting both kinds of functions on microcontrollers, the non-health and safety functions can potentially impact the health and safety of the vehicle if they have bugs, consume too many resources, etc.

In our automotive architecture we’d like to take these non-health and safety functions and hoist them out of the microcontrollers and into the general purpose compute device. Unfortunately, these functions cannot afford the performance impact of containerization, so there needs to be a second zone in the untrusted zone that is for un-containerized, but also untrusted, functions.

Putting this altogether, we end up with a hardware and software architecture stack like the one below. Hardware provides metadata and security information that enables a simplified digital representation of the vehicle to be instantiated. This digital representation interface is common
across all makes and models of vehicles. The trusted zone provides the “heavy lifting” for creating and managing untrusted workloads. Untrusted workloads live in fully isolated containers whenever possible, but there is accommodation for those workloads that need special handling.

Fig. 4: Complete architecture

**Cloud technology in the vehicle**

In the previous section we described a single node system. However, as complexity in the vehicle grows it will transition further into a data center on wheels. Future automotive architecture will include compute clusters of multiple compute units each with specialized co-processing capabilities.

The usage of an Edge Compute Fabric (ECF) with container orchestration technology can increase reliability and simplify the management of vehicle workloads. With the ECF, the control plane is responsible to manage all workloads and communication in the system. The nodes run on the vehicle controllers and support several containerized modules.
Workloads can be developed and deployed from the cloud to be delivered to the vehicle. The edge compute fabric will look at workload attributes to determine if the cluster includes a compute unit optimized for that workload. If the workload is generic, then the fabric can use a load balancing algorithm to determine which compute unit will handle the workload.

With this approach, it is possible to increase efficiency and reliability at the expense of some additional software complexity.

**Conclusion**

The future of automotive software is one of customer centricity and constant change. Automotive companies will add new functions constantly to align with customer expectations and unlock new revenue streams. The upgrade to modern hardware with increased compute, storage and memory capabilities coupled with general-purpose graphical processing units is already underway. Unfortunately, it is not enough: It is necessary to also rethink the in-vehicle architecture and development processes. It is also useful to look at adjacent industries and study how they solved similar challenges. A good example is how Microsoft addressed the transition from standalone networked computers to an always-connected, updateable operating system with a rich ecosystem of partners. The usage of new cloud-inspired
technologies such as containerization and digital twins is useful to increase agility while keeping a safety-first approach.

Works Cited


