Securing the Automobile: a Comprehensive Approach

Lee Pike, Jamey Sharp, Mark Tullsen, Patrick C. Hickey, James Bielman
Galois, Inc.
{leepike|jamey|tullsen|pat|jamesjb}@galois.com

December 20, 2015

Abstract

Previous research has demonstrated the software and network vulnerabilities that exist in modern automobiles. We present a rejoinder to that work here, discussing approaches to secure the modern automobile. The approach is comprehensive, taking into account static assurance, dynamic assurance, software, networks, while considering the constraints of the automotive industry.

1 Introduction

In 2010 and 2011, research performed jointly between the University of California, San Diego and the University of Washington[1] convincingly shows that modern cars can be completely compromised remotely by exploiting software-based vulnerabilities [1, 2]. In modern cars there is a lot of software to hack: estimates suggest that a luxury car contains 10s of millions of lines of code, over 100MB of binary code, executing on 50-70 Electronic Control Units (ECUs), each containing independent computer systems built from micro-controllers, digital signal processors, multimedia processors, or application processors. A modern economy car might contain 25+ ECUs, and that number continues to grow.

The key enabler of the attacks described by the UCSD/UW team was the ease of gaining access to at least one of the numerous data buses on the automobile. These buses allow the ECUs to coordinate with each other—e.g., so the braking system can interact with the engine controllers to provide better control. Some ECUs act as data-bus bridges and are able to broadcast on multiple buses. In the automobiles analyzed, *every* ECU is transitively connected to every other ECU via data-bus or data-bus bridges. This highly connected architecture is driven by complex interactions required for safety or desired for comfort. For example, the door lock system must know whether the airbags have deployed so it can automatically unlock all doors to make escape easier, and the entertainment system must know the vehicle's speed so it can raise audio volume to compensate for wind and road noise.

The UCSD/UW team found multiple attack vectors through which to launch an attack against the automobile. These include a malicious audio track played in the CD player, malicious Bluetooth access to the entertainment system, and malicious cellular access to the telematics system. Other vectors were also considered, such as hacking a mechanic's diagnostics tool remotely.

In each attack, some interface vulnerability is initially exploited. These include, for example, some combination of brute-force guessing short PIN numbers (e.g., for Bluetooth), exploiting buffer overflows in low-level networking code, shell-code injections, and automated firmware updates. Once access to the a data-bus was gained, further access is obtained by reprogramming other ECUs over the data-bus, using mechanisms designed so that a mechanic performing software updates does not require physical access to individual ECUs.

[1] Hereafter referred to as the UCSD/UW team.

Contributions While the UCSD/UW team has demonstrated the software vulnerabilities of the modern automobile, there has not been a comprehensive set of recommendations outlining how to mitigate them; that is the purpose of the present paper. Our goal is to provide a holistic perspective, focusing on security at different levels of abstraction (i.e., ECUs, the architecture, etc.) and using a broad set of approaches.

We assume that the reader is familiar at a high level with the kinds of vulnerabilities that researchers have reported. We propose mitigations which we believe automobile manufacturers can begin implementing today for incremental improvements in security, as well as longer-term solutions to address deeper challenges.

Some security goals align with broader safety demands. Section 2 discusses areas where security can be thought of as an extreme form of the safety requirements which car manufacturers already pursue.

Any security recommendations which do not address the realities of business requirements, usability, and practicality are doomed to fail. In Section 3, we discuss the constraints that limit which security recommendations are feasible.

The bulk of our recommendations apply at the level of software found in individual ECUs, and can be found in Section 4. Recommendations at this level can generally be applied by individual subcontractors without coordination across the supply chain, making them more practical for immediate application. We consider both static and dynamic (runtime) assurances.

However, some security issues can only be addressed at the level of the entire system architecture, as discussed in Section 5. Since the product development cycle lasts several years in the automotive industry, significant architectural changes are not expected to appear until design cycles beginning today hit the market.

Finally, some security concerns must be addressed in supply chain management, as we discuss in Section 6.

One contribution of [1, 2] is to elucidate the breadth of the *attack surface* of a modern automobile, which is broader than one might expect (or hope) due to the interconnection of buses and the presence of many external interfaces (USB, Bluetooth, etc.). Some notable attack vectors here:

- The OBD-II port, which transitively includes internet-enabled computers connected to the port for diagnostics purposes.
- The Bluetooth port, is reachable outside the physical confines of the automobile.
- The MP3-CD player among a host of external interfaces: refer to [2] and also Section 4.5.

Since the attacker has numerous channels or attack vectors (the automobile's attack surface) to "enter the system," all that is remaining for the attacker is to find some software vulnerability that is accessible via any of these vectors. There are dozens of common software defects which could be security vulnerabilities; some of the more common of these are

- Input validation errors such as format string attacks and SQL injection;
- Memory safety violations such as buffer overflows and dangling pointers.
- Privilege confusion or privilege escalation bugs.

2 Safety Versus Security

Throughout this paper, we discuss the tensions between safety and security. For example, in Section 5, we describe why data buses might be connected for safety reasons while separation provides better integrity and confidentiality guarantees. On other dimensions, however, safety and security are in alignment, and in many cases security can be thought of as an extreme form of safety. We discuss the connections between safety and security below with respect to the bus architecture and ECU software, respectively.

Architectural Safety and Security In fault-tolerance, there is an unattributed saying, "Time turns the improbable to the inevitable." Another way of putting this is that a sufficiently improbable fault is indistinguishable from a malicious attacker. In distributed systems design, a *Byzantine* fault model is one model corresponding to an attacker [3]. A Byzantine fault is one in which a node that is designed to

broadcast the same message to two recipients in fact sends arbitrarily different messages to each. Byzantine faults are observed in nature [4], and data buses have been designed to protect against them [5]. By using a data-bus architecture with sufficiently strong fault-tolerance guarantees, one gets some additional integrity protections, for free.

In particular, the CAN bus, one of the primary buses used in modern automobiles, is not particularly safe or secure. There is no protection against "babbling idiots" on the bus, in which a node denies service to other nodes, either because of a fault or in a malicious denial of service. Furthermore, it is difficult for nodes to reliably diagnose timing faults in other nodes since there is no notion of clock synchronization of the nodes and the bus is not time-divisioned. Finally, since the bus is not replicated, (potentially malicious) line noise can cause Byzantine faults among recipients that cannot be detected by error detection mechanisms like CRCs [6].

Software Safety and Security Automobiles are designed to be safety critical, and automotive safety integrity levels (ASIL) classifications, defined by ISO 26262, provides guidelines for assessing the risk of hazards if particular functions fail [7]. Four levels of severity are defined, from level A to level D (highest). The ASIL classification corresponds in spirit to similar classifications in commercial aviation [8]. At higher ASIL levels, additional guidance is given for the software development process to decrease the chances of introducing a flaw.

However, when calculating the probability of failure in a hazard analysis, software is assumed to be fault-free [9]. Failure probabilities are typically based on the failure rate of hardware components under anticipated environmental conditions, assuming software functions without flaw. Highly-improbable systemic software flaws—when multiplied across the automotive fleet size the software is deployed in and multiplied against the number of operational hours each automobile will have in its lifetime—can easily lead to catastrophic failure being more probable than not.

3 Automotive Business Constraints

Business constraints can be in tension with developing secure systems. Automotive constraints are particularly antagonistic. We highlight the following constraints:

- Part cost: The automotive industry is extremely sensitive to part cost.
- Size and weight: Solutions must be tempered by the increase in size or weight. For example, replacing a CAN network (line topology) with an ethernet network (star topology with a central switch) may be infeasible due to the extra wiring required.
- Legacy integration: The automobile industry often depends on long component lifetimes to keep costs down, so a new design may have to integrate with legacy components. Approaches that require major architectural changes, such as a change in bus technology, may be delayed or ruled out in order to remain compatible with legacy components.
- Memory constraints: Cost pressures require that ECUs use the least expensive components which can do the job, and particularly in small microcontroller based ECUs, memory is the most costly part of a component. Software for these micro-controller ECUs is designed for the smallest memory footprint possible, and security approaches which use large amounts of memory may be ruled out on the basis of cost.
- Timing requirements: Many ECUs perform tasks which have fixed real-time deadlines, (i.e. the time between receiving a command and carrying out that command), and these deadlines are often safety-critical. Security measures which may prevent a program from meeting timing requirements are ruled out on the basis of safety.
- Standardization: A single manufacturer cannot afford to break away from industry standards. Keeping supply chain costs low requires leveraging suppliers making similar parts for multiple manufacturers.

Some recommendations we make conflict with the constraints above. On the other hand, the goal line for some of today's technical constraints will move. For example, as processor manufacturers retire old components, some processors will be upgraded to modern alternatives which offer superior security features for free—e.g., modern micro-controllers offer memory protection, a feature which can be used to ensure software components do not interfere with each other.

The non-technical constraints are more difficult. One of the largest hindrances to security is suppliers providing components as a "black box". Parts are provided by suppliers with only an assurance that, in benign operating conditions, they function according to specification. The manufacturer provides few or no security-oriented requirements, such as a spec for how parts should function in adversarial operating conditions, and the supplier, in order to protect the intellectual property in their software, does not provide any visibility into the software used in the part, preventing the software from being assessed, modified, or instrumented in order to increase security.

We envision some changes to the status quo that might allow automobiles to become more secure.

- Independent evidence: Manufacturers should require evidence that security objectives are met by a supplier's software. If a manufacturer cannot analyze the software directly, it can require that the supplier provide the evidence. The evidence might be in the form of test results, documentation, or signed guarantees. A third-party analyst can also corroborate a supplier's claims.
- Collaboration: Although a supplier may not want to provide a manufacturer with a full copy of each part's software, suppliers might benefit from allowing a manufacturer to embed their own engineers within their teams. Joint teams between suppliers and manufacturers can contribute to and audit both the source code and the software development process while reducing risks to the supplier's trade secrets.
- Open software: Suppliers might be convinced that opening access to their software is a competitive advantage—e.g., a "Red Hat model of business" [10]. Open software makes it possible for others to analyze and improve the software. In a sense more limited than open-source, suppliers might find ways to offer to provide manufacturers with their software source, and use patents and licensing restrictions to protect their trade secrets.
- Liability: Finally, there is precedent for holding automotive manufacturers liable for harm caused by their software [11]. Business impediments to security may naturally be reduced. In particular, manufacturers may pass on liability to suppliers, giving suppliers a much bigger stake in ensuring software security.

Ultimately, it may be very difficult to ensure software quality with the current supplier model. As noted in the UCSD/UW research,

> ... while this outsourcing process might have been appropriate for purely mechanical systems, it is no longer appropriate for digital systems that have the potential for remote compromise [2].

4 Assuring the ECUs

A modern vehicle contains dozens of embedded computers distributed throughout the vehicle. Some, like the head unit, contain a laptop-grade processor, a full operating system stack, and a user interface. Others, such as those for seat positioning, might be 8-bit (or smaller) microcontrollers with just a few kilobytes of memory. Others fall between these extremes.

Attacking a modern ECU involves modifying its program (firmware), modifying the data that program is operating over, or modifying its hardware implementation. The primary attack vectors for modifying a program or its data are flaws–or misued features–in the program itself.

In this section, we cover approaches to improve software development, testing, formal verification, and runtime assurance, particularly borrowing on high-assurance development in other domains. We discuss hardware modifications in Section 6, where we discuss supply chain issues.

4.1 Software Development

The *Pareto Rule* applies to software quality: 20% of the effort is responsible for 80% of the quality. The point is that by following software development practices that are well-known and low-overhead, many security vulnerabilities can be eliminated early in the design and implementation process. Still, the automotive industry has often ignored the "low-hanging fruit" for improving software quality [11], such as using version control, unit testing, integrated testing, and code reviews, which are already recommended by the MISRA Software Guidelines [12].

Below, we describe additional approaches to help improve software quality.

Coding Standards Coding standards can help improve software quality. For C software development, the MISRA C standard [12] is common in the automotive industry. (However, it is not universally followed, [11].) About a dozen static analysis tools can check for MISRA conformance. A much simpler but compatible coding guideline with ten rules is proposed by researchers at JPL [13].

Static Analysis Automated software static analyzers analyze the source code of a program and alert the user to possible flaws. Popular commercially available static analyzer tools include Code Sonar [14], Coverity [15], and Polyspace [16]. Some tools are sound, meaning they should not produce false negatives. For scalability and to reduce false positives, some tools are unsound. Software written according to guidelines like MISRA can improve the performance and usefulness of static analysis tools.

While static analysis is a powerful tool, we caution that it can also lead to a false sense of security. Toyota used static analysis tools on their acceleration ECU software, which was found by third parties to have many quality issues that may have been solved by better following software engineering best practices [11]. Static analysis tools produce false positives so many false positives that it is difficult to discern the wheat from the chaff. Also, static analysis cannot be relied upon to uncover domain-specific bugs automatically. Indeed, some properties cannot be encoded as code-level assertions.

Memory-Safe Programming Approximately half of the vulnerabilities exploited by UCSD/UW are at their heart the result of buffer overflows [2]. Buffer overflows as a security vulnerability have been known since the 70's, but due to unsafe languages, still exist. Buffer overflows are a particular example of a memory-safety violation, which itself is an example of undefined behavior.

Coding standards and static analysis are mostly targeted at preventing and discovering, respectively, undefined behavior (e.g., caused by a buffer overflow) resulting from using "unsafe" programming languages like C or C++. C/C++ is commonly used for ECU programming. Why? There are four reasons:

- Legacy concerns: Existing code in C/C++ represents a significant investment by software authors.
- Space: Most high-level languages have a much larger memory footprint than C/C++ programs of comprarable functionality, and memory size is a major cost concern.
- Timing: ECUs typically have hard real-time deadlines. Programs written in C/C++ can typically be written so that their execution time is determinstic and bounded.
- Low-level programming: Finally, without restrictions on pointer use and type-casting, C/C++ naturally supports the idioms required for writing device drivers that interaction with hardware.

Safe-C languages are languages that are advertised to be safer than C, and that require a small (comparable to C) runtime system. These languages are designed to replace C for some subset of applications commonly written in C because of C's advantages mentioned above.

Two modern *safe-C* languages are Rust [17] and Ivory [18]. Rust is focused on two primary goals–concurrency and memory-safety–while still providing to the developer good performance and low-level control of memory. The language guarantees memory-safety through static type checking rather than a run-

time system. Therefore, Rust programs are nearly as fast and predictable with as small memory footprint as C programs.

Ivory was developed by the authors to support the DARPA HACMS program [19]. Ivory is a secure alternative to C/C++ in which memory-safety errors are impossible and that supports a variety of verification tools.

4.2 Testing

Testing is the primary means in industry to provide software assurance. We will not address testing in more detail except to point out two classes of testing that are highly effective but less often used, *fuzz testing* and *property-based testing*.

Fuzz Testing A class of testing which has been successfully used for security-critical systems is *fuzz testing*, in which inputs that do not correspond to the documented application interface are generated. Fuzz testing is particularly effective for discovering bugs caused by insufficiently sanitized user input [20].

Property-Based Testing *Property-based testing* refers to a testing approach in which test-cases are (mostly) automatically derived from the property specification, running tests until a specific test-case fails or the tester stops. Property-based testing tools generally use simple test-cases initially, iteratively generating more elaborate tests.

QuickCheck was one of the first property-based testing tools developed [21]. QuickCheck has been used in the automotive industry. For example, Volvo is using QuickCheck to test conformance against the AUTOSAR 4.0 standard [22].

4.3 Formal Verification

While testing provides partial assurance about the actual artifact to be fielded (since a failing test vector can always be missed), formal verification provides complete assurance about a model of the system. With testing, the designer's worry is, "Have I tested enough?" With formal verification, the worry is instead, "Is my model's fidelity accurate enough?".

Static analysis, discussed in Section 4.1, is a form of formal verification in which constructing the model is performed by the static analyzer itself. Because model construction is automated, making it domain-specific is difficult, so static analysis usually focuses on modeling the semantics of a programming language. Static analyzers are therefore low-level and cannot generally deal with systems (e.g., networked devices), domain-specific properties, or even complex software (e.g., concurrency).

Formal verification then requires a two-step approach: build a model, and then verify it. Fortunately, for automotive systems, models may already exist for use in simulation.

At the model level, a potentially useful class of formal verification tools are model-checkers, which are particularly useful for automating the analysis of concurrent or distributed systems [23].

Other approaches include interactive theorem proving. Theorem proving is more general but not very automated, requiring significant user expertise. Theorem provers have been used to verify real-time data bus designs, though [24].

Glue Code Generation Glue code is software that interfaces between software subcomponents. Glue code is responsible for integrating a collection of reusable components into a complete system. It might be responsible for modifying data formatting as it is passed between components, or translating new software interfaces to work with legacy components.

Glue code is conceptually simple, but it is often where errors occur because correctness relies on understanding both the requirements and the assumptions of all of the software components it touches. Flaws in glue code are responsible for multiple published security attacks on automobiles, including Bluetooth usage, diagnostic "PassThru" systems, and even the audio system [2].

Ideally, glue code can be generated from specifications rather than written by-hand. In general, a top level specification which described the assumptions and requirements of individual components is good engineering practice, and glue code generated

from such a specification can be a major part in ensuring the specification of the components matches the implemention of components. This approach can also reduce engineering cost by reducing the time to develop and test of tedious boilerplate [18].

4.4 Runtime Assurance

In this section, we describe approaches to ensure a program's execution meets a specification. These techniques are distinct from testing or other analysis techniques performed at design time, as described the preceding section.

System Specialization In the UCSD/UW analysis, some middleware contained a full installation of an operating system (e.g., Linux), complete with standard root-level networking tools. There was no need for these tools to exist on the system, but they were leveraged during the study to simplify the attacker's analysis of other parts of the system and to simplify software attacks. As noted by Checkoway *et al.*:

> Finally, a number of the exploits we developed were also facilitated by the services included in several units. For example, we made extensive use of `telnetd`, `ftp`, and `vi`, which were installed on the PassThru and telematics devices. There is no reason for these extraneous binaries to exist in shipping ECUs, and they should be removed before deployment, as they make it easier to exploit additional connectivity to the platform [2].

The lesson is that only those tools required should be installed.

Taken to its conclusion, operating systems researchers have been developing a *unikernel* approach in which an operating system and drivers is developed as a set of specialized libraries, and depending on the applications running on it, only the required libraries are linked in. Two virtual machines implementing the unikernel approach are the HaLVM, developed at Galois, Inc. [25]; and Mirage, developed at Cambridge University [26]. Tools like HaLVM and Mirage are particularly relevant for securing systems built on modern application processors, such as the entertainment system ECU.

Data Integrity *Measurement* and *attestation* (M&A) are, respectively, approaches to check the value of data, including the executable, and then prove to a third-party that the values are as expected [27]. M&A often assumes the existence of special hardware (like a Trusted Platform Module) to provide a root of trust [28], which may exist on small microcontrollers; lighter-weight solutions have been proposed in the automotive industry [29].

M&A is particularly relevant to ensure that ECUs are only executing binaries which have been endorsed by the manufacturer. Reflashing ECUs with maliciously modified binaries was a key element in a number of the attacks described in the UCSD/UW work.

Runtime Verification *Runtime verification* (RV) is an active research field that marries formal verification and testing. The idea is to take a high-level specification about program behavior and instrument a program's internals or output to check for conformance against the property.

The challenges associated with RV include how to instrument a program to check all control-paths that are relevant to the property, while ensuring that the instrumentation does not adversely change the behavior of the program (particularly the nonfunctional behaviors, such as timing and memory usage). Previous work by the authors, addressed these problems in the context of hard real-time avionics [30].

Software Fault Containment Regions The concept of *fault-containment regions* (FCRs) is fundamental to fault-tolerant system design [31]. The idea is to design architectural regions that contain classes of faults. For example, separate cabinets with separate power supplies suffer power failures independently. Replicated ECUs for critical applications provide some level of fault-containment. However, an argument that hardware FCRs provide redundancy assumes there is no common-mode failure in any repli-

cated software. Thus, software vulnerabilities can undermine safety arguments, as described in Section 2.

Similarly, FCRs make sense for software as well as hardware. For example, memory isolation is particularly important so that memory safety errors cannot propagate between programs. Small ECUs typically lack memory management units.

Moreover, software functions follow the principle of least privilege and share only the data required by other functions. In particular, diagnostic and debugging information should be separated from nominal payloads and be enabled only in specific modes.

4.5 Sanitizing Inputs

Automobiles have a variety of interfaces on which they accept input. From a security perspective, every input interface must be considered completely under the control of an attacker. All input interfaces offer some attack surface for an attacker to subvert assumptions of software which interacts with that interface. The key to defending against attacks on untrusted interfaces is for the software to assume nothing about the input. This means it must have valid modes for handling any input.

Let us inspect a number of common interfaces and discuss vectors of attacks on each.

Radio Systems Ordinary FM and AM audio decoding is straightforward enough. However, new standards, such as Radio Broadcast Data System (RBDS), provide machine readable information outside the audible band of an FM radio signal [32]. RBDS may provide information relevant to the radio tuner itself, the audio system, and traffic information relevant to a variety of other systems in the car. This relatively simple input may have a broad reach throughout the car's systems, interacting with tens of thousands of lines of code just to manage string manipulation and character encoding.

Other digital radio standards, such as HD Radio and satellite radio, have problems along the same lines, but are considerably more complex than RBDS.

Media Systems Nearly all cars support playback of audio provided by the user in digital format. These systems are required to deal with an enormous amount of complexity. To go from a user-provided MP3-CD, SD card, or USB drive to an audio stream, the input must interact with many hundreds of thousands of lines of code throughout a stack of software components, starting with low level device drivers, moving up through filesystems, metadata parsers, audio and video decoders, and display systems.

Additional complexity ripe for attack includes decoders to interact with iPod, iPhone, or Android phones over USB, and subsystems that classify or recognize a newly attached device. Attacks have been demonstrated in the wild against these kinds of targets [33].

Telematics Systems Today's high-end cars have complex telematics units connected to the cell phone network. These are used for emergency services as well as day-to-day conveniences. Any vulnerabilities in these systems are particularly hazardous since they may be remotely exploitable by anyone with access to the phone network, and because these telematics units must have access to critical systems for emergency services purposes. Unsurprisingly, exploitable vulnerabilities have been demonstrated in telematics systems [2].

Wireless Key Systems The most widely-understood class of attacks among the general population of car owners: If an attacker can unlock your car without possessing your keys, that is a violation of trust that consumers recognize at a visceral level. Unlike the other categories, the attacks which have been demonstrated have not typically been due to software implementation bugs. They've either been caused by inadequate cryptographic protocol design, or physical threats such as relay attacks [34]. Still, even perfectly-applied cryptography and physical defenses would be useless if a buffer overrun allowed them to be bypassed.

Vehicle to Vehicle Communication Vehicle-to-vehicle (V2V) systems, under proposal by the Department of Transportation [35], do not exist in commodity vehicles and have not been subject to vulnera-

bility analyses, but they will present additional security challenges and vulnerabilities [36]. V2V involves a complex security infrastructure, aimed at providing authentication while providing anonymity to users. Implementations will consequently require complex software and will likely provide another attack vector for the vehicle.

5 Architectural Assurance

It is common in modern vehicles to use multiple buses to connect different ECUs together, for several reasons. Some links can be low-bandwidth, in which case LIN is preferred to reduce cost. At the other extreme, the amount of communication over high-bandwidth links has been growing beyond the capacity of a single CAN bus. Very high bandwidth flows are migrating to Ethernet or FlexRay. Others are being split onto dedicated CAN buses.

There are good security reasons to use multiple buses as well, although as we'll see, there are trade-offs that must be considered.

Ideally, nodes which do not need to communicate with each other should be partitioned onto separate buses to limit damage if other countermeasures fail. It's especially important that safety-critical nodes should be on a dedicated bus, not shared with anything less critical.

Sometimes, two buses must be bridged, perhaps because data produced by a critical node is needed by a low-criticality component such as the entertainment system. In that case, each bridge should only forward messages which have been explicitly permitted by the system design, and should rate-limit those messages to keep denial-of-service attacks on one bus from affecting others.

However, each additional bus adds to wiring costs; separation adds to architectural complexity, which has an engineering cost; and introducing bridges between buses adds to electronics costs.

Furthermore, there are often non-obvious reasons why seemingly-unrelated nodes need to communicate with each other. For example, door locks might normally be a low-criticality interface, while detecting a car crash is a highly safety-critical process—but there may be a requirement that when a car crash occurs, the doors automatically unlock. As a result, there is a complicated trade-off at the architectural level between safety, reliability, and security.

Challenges The obvious way to guarantee the confidentiality and authorization of data in transit is with cryptography. However, there are several significant challenges to adding cryptographic authentication to ECU communications today.

First, most messages of interest are delivered over CAN, which in the current technology generation limits each message to 8 bytes in length. Since 8 bytes is the smallest a cryptographic signature can be while still having reasonable security, any authentication scheme on current CAN buses requires a message fragmentation protocol.

Since some communications are already fragmented–notably dealer diagnostic commands–these are a natural first target for strong cryptography. In addition, future technology generations will relax these restrictions. Ethernet frames can be up to 1,500 bytes on standard hardware, and FlexRay frames can be up to 254 bytes long. If the CAN-FD extension to the CAN standard is adopted by the automotive industry, compliant CAN nodes will be able to exchange frames up to 64 bytes long [37].

Another challenge is the cost of hardware security modules (HSMs). An attacker may have full physical access to the electronics in the vehicle, and may have financial incentive to invest in expensive equipment. As a result, tamper-resistant key storage would be preferable; unfortunately, this could add significantly to parts cost.

Two factors mitigate this problem. One is that that cost of HSMs is falling to the point that some automotive manufacturers are considering them for cars in the next few years [29]. The other factor is that it is not necessary to prevent tampering or key extraction given physical access to the vehicle, so long as extracting the key from one car doesn't make any other cars vulnerable. This requirement can be met given a public-key infrastructure, where each component is assigned a random key-pair signed by the manufacturer, and then key exchange computes ses-

sion keys which are not saved across power cycles.

For messages which are time-critical, performance is a third challenge. Cryptographic functions take time to compute, especially on a microcontroller that doesn't have hardware acceleration for the cryptographic primitives. Here, too, the decreasing cost of HSMs will make hardware accelerated crypto available to more parts of the car over time. Still, messages which are not time-critical should be the first targets for security improvements.

Recommendations The first step is to use strong cryptography for messages intended for dealers' use in the field. This includes commands for firmware updates and other vendor-specific diagnostics. Many manufacturers already do some portion of this, although the implementation details should be reviewed by cryptography experts and tested by a team of penetration testing experts.

Diagnostic messages are the best case for each of the challenges that automotive cryptography faces: They are often already longer than one CAN message, so they already require fragmentation; they typically aren't addressed to the smallest microcontrollers; and they are not normally time-sensitive. These messages are also critical to protect, because if you can't trust that the firmware loaded on an ECU is authentic, that undermines other efforts to harden ECU security.

Given additional resources, the next step is to add authentication to critical messages sent during normal vehicle operation, such as:

- commands to brakes, throttle, steering, locks
- status for speedometer, fuel gauge
- data from lateral acceleration sensor to airbags

OBD-II Port Independent of the architecture of the internal data busses, the government-mandated OBD-II port provides external access to most ECUs. While access to the OBD-II port requires physical access to the vehicle, it is security-critical (and even safety-critical) and should be given careful attention.

Only the specified diagnostic messages should be allowed to be transmitted to/from the OBD-II port. This includes not only the kind of message, but in what vehicle state they are administered. If a command that is useful for diagnostics would be dangerous if issued at highway speeds, then the command should only be recognized in a special system diagnostic mode, which should be authenticated.

Head Unit System Isolation The head unit, including the telematics and entertainment systems, is at the highest risk of having an exploitable vulnerability. At the same time, it is the easiest to compartmentalize, since it is centrally located within the vehicle, minimizing extra wiring, and it already uses relatively high-end CPUs with a variety of security features and support for higher-cost buses like Ethernet.

The head unit should be placed on an untrusted bus, which might be CAN or Ethernet, together with a separate ECU to bridge and filter that bus to the critical buses in order to forward necessary data. Ideally, these units should also run their software under a trusted virtualization hypervisor with a small filter driver that prevents the node from sending or receiving traffic it is not supposed to exchange (see the discussion on system specialization in Section 4.4). Notably, they should not be permitted to send diagnostic commands; in fact, the bar for allowing these components to send any messages at all to the rest of the vehicle should be set very high.

Some interfaces within the vehicle must be trusted in order for systems to work properly. For instance, the brake controller must trust inputs which indicate whether to apply the brake or not. Since the brake controller is receiving that input from an external interface which may have been compromised, the brake controller should authenticate the input.

Currently, messages broadcast onto automobile CAN and LIN buses have no way of authenticating their origin. Due to the broadcast nature of these buses, there is no protection at the hardware level from an attacker adding an additional device to these buses to send maliciously crafted messages. Worse, the government-mandated OBD-II port makes execution of this kind of attack easy.

In addition, there is no protection against reprogramming existing devices connected to these buses

to send maliciously crafted messages, and, as shown in the UCSD/UW work, the dealer's equipment could even be subverted to attack known vulnerabilities in each car brought into the shop.

To avoid these issues, important messages between ECUs should be cryptographically authenticated. It may also be desirable to encrypt them, to make it difficult for an attacker to determine when to execute an attack based on the state of the vehicle. Conveniently, with modern authenticated-encryption block cipher modes like Galois Counter Mode (GCM), encryption comes free with the implementation of authentication [38].

Finally, in an ideal world, manufacturers would add authentication and encryption to most or all messages. This is expected to be a long-term effort because it requires coordination between many different vendors throughout the supply chain for each car manufacturer.

6 Assuring the Supply Chain

Recent years have shown a growing concern over the robustness of supply chains, in particular those used to obtain critical "microelectronics supplies for defense, national infrastructure, and intelligence applications," [39]. One concern is over the possibility of introducing a *hardware Trojan*: circuitry that lies dormant (e.g., often doing nothing except monitoring external stimuli) until it observes a *trigger condition*, at which point it performs an attack.

The attack surfaces facilitated by hardware Trojans are numerous: proof-of-concept attacks have demonstrated troublesome capabilities including data exfiltration [40], gradual degradation or destruction of system components via malicious process parameterization [41], privilege escalation and credential capture [42], and injection of modified firmware [42]. Many proposed Trojans suggest ways to realize "kill switch" functionality [43], wherein deployed systems (e.g., UAVs or radar systems) can be shut down or behave maliciously when a trigger condition is received (e.g., a trigger value obtained via an adversary's transmission device).

In 2005, the *Defense Science Board Task Force on High Performance Microchip Supply* performed a study [39] of the threat and made various recommendations for mitigating the problem. This study was done in the context of "defense, national infrastructure, and intelligence applications," but the same vulnerabilities exist in the commercial sector, and *in the case of automobiles*, the impact of exploits may also extend to the loss of human life.

In the commercial space (in contrast to the defense space), it may be problematic for the manufacturer to acquire from component vendors information that may expose, to various degrees, their intellectual property: e.g., evidence that a component is free of vulnerabilities or that it was developed with a certain process. We touched on this problem in Section 3: we suggested (under *independent evidence*) how the supplier might provide test results, documentation, or third party guarantees.

The following are approaches to secure the supply chain:

1. secure the process of developing and delivering components;
2. remove vulnerabilities from components (hardening components against threats);
3. apply the principle of least privilege (limiting the capabilities of the attacker from a compromised component);
4. detect malicious circuitry or code in the components;
5. add run-time mechanisms for thwarting and/or detecting malicious behavior.

All of these defenses are orthogonal and could be used profitably in conjunction, thus achieving Defense in Depth. Due to space considerations, we discuss only the last two, which are areas of active research, in more depth.

A large body of work has been produced in the last few years that describe various ways of tackling the detection problem of hardware or firmware Trojans. This work ranges from high-level taxonomical descriptions of Trojans and detection techniques [44], to statistical techniques [45], to specific detection methods such as power calibration or gate-level behavior characterization [46]. A variety of "fingerprinting" methods have been proposed to allow post-fab identification of modified circuitry, either via path delays

in golden circuits [47] or by the collection and provable correspondence of data extracted from various side channels [48]. Finally, extensions to existing testing methodologies propose new test vector generation strategies based on theoretical models of anticipated Trojan behavior and Trojan insertion methods [49].

Another class of countermeasures is to add diversity to our systems. Diversity at the network communications level could inhibit Trojans from being activated or inhibit their actions if triggered. Activating a Trojan requires the adversary to be able to deliver a trigger to some data path that the Trojan can observe. By introducing some degree of obfuscation in the hardware or in the software, the attacker's assumptions are violated, potentially preventing the trigger from being delivered or recognized.

Synthetic diversity at the code level has been proposed as a countermeasure to code vulnerability exploits: by creating multiple–functionally equivalent–variations of the code, we make it more difficult for the attacker to create an attack that works on every variation of the code [50].

7 Conclusions

Drivers' sense of control over their vehicles is psychologically critical. Having that sense of control threatened by a security vulnerability gets substantial negative media and consumer attention. If any of these vulnerabilities are exploited and lead to property damage, injury, or death, the legal liabilities are also a significant concern. Naturally, the automobile industry understands this, but the gap between security researchers' recommendations and the practical constraints that the manufacturers operate under have led to missed opportunities for improvement.

In this paper, we have presented the constraints that have limited automotive adoption of the best-known methods in computer security, as well as our recommendations for solutions and mitigations that satisfy those constraints. Our recommendations cover the range from individual microcontrollers to system architecture level design questions and all the way through supply chain management concerns.

It is our hope that the automotive industry at all levels can find easy process enhancements in this report which they can apply today for incremental improvements in security, and more significant changes which they can apply over time to resist substantially more advanced threats. In addition, we intend that security researchers can take from this report some guidance on fruitful directions for research results that will be applicable to the auto industry's needs.

Acknowledgments

This work was partially supported by DARPA contract HR0011-14-C-0113. All opinions expressed herein are our own.

References

[1] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile," in *IEEE Symposium on Security and Privacy*, 2010.

[2] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive experimental analyses of automotive attack surfaces," in *USENIX Security*, 2011.

[3] L. Pike, J. Maddalon, P. Miner, and A. Geser, "Abstractions for fault-tolerant distributed system verification," in *Theorem Proving in Higher Order Logics (TPHOLs)*, ser. LNCS, vol. 3223. Springer, 2004, pp. 257–270.

[4] K. Driscoll, B. Hall, H. Sivencrona, and P. Zumsteg, "Byzantine fault tolerance, from theory to reality," in *Computer Safety, Reliability, and Security*, ser. LNCS, vol. 2788. Springer, 2003, pp. 235–248.

[5] P. Miner, A. Geser, L. Pike, and J. Maddalon, "A unified fault-tolerance protocol," in *Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems (FORMATS-FTRTFT)*, ser. LNCS, vol. 3253. Springer, 2004, pp. 167–182.

[6] M. Paulitsch, J. Morris, B. Hall, K. Driscoll, E. Latronico, and P. Koopman, "Coverage and the use of cyclic redundancy codes in ultra-dependable systems," in *2005 International Conference on Dependable Systems and Networks (DSN)*, 2005, pp. 346–355.

[7] *Development Guidelines for Vehicle Based Software.* MISRA, 1994, iSBN 0 9524156 0 7. Out of print. Available from www.misra.org.uk.

[8] Radio Technical Commission for Aeronautics (RTCA), "DO-178B: Software considerations in airborne systems and equipment," 2011, http://www.rtca.org/onlinecart/product.cfm?id=501.

[9] N. G. Leveson, *Safeware: System Safety and Computers*. New York, NY, USA: ACM, 1995.

[10] R. Young, W. Rohm, and R. H. Inc., *Under the Radar: How Red Hat Changed the Software Business–and Took Microsoft by Surprise*. Coriolis Group, 1999.

[11] P. Koopman, "A case study of toyota unintended acceleration and software safety," Public seminar, September 2014, available at http://www.slideshare.net/PhilipKoopman/toyota-unintended-acceleration?ref=http://betterembsw.blogspot.com/.

[12] *Guidelines for the Use of the C Language in Critical Systems*. MISRA, 2004, iSBN 0 9524156 2 3 (paperback). Available at http://193.35.217.33/Buyonline/tabid/58/Default.aspx.

[13] G. Holzman, "The power of 10: Rules for developing safety-critical code," *Computer*, vol. 39, no. 6, pp. 95–97, Jun. 2006.

[14] "Code sonar," Website, available at http://www.grammatech.com/codesonar.

[15] "Coverity," Website, available at http://www.coverity.com/.

[16] "MathWorks polyspace," http://www.mathworks.com/discovery/static-code-analysis.html, May 2015.

[17] N. D. Matsakis and F. S. Klock, II, "The rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14. ACM, 2014, pp. 103–104.

[18] P. C. Hickey, L. Pike, T. Elliott, J. Bielman, and J. Launchbury, "Building embedded systems with embedded DSLs," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '14. ACM, 2014, pp. 3–9.

[19] "High-assurance cyber military systems," Website, available at http://www.darpa.mil/Our_Work/I2O/Programs/High-Assurance_Cyber_Military_Systems_%28HACMS%29.aspx.

[20] A. Takanen, J. DeMott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.

[21] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00. ACM, 2000, pp. 268–279.

[22] "Checksum property for AUTOSAR," Website, December 2014, available at http://www.quviq.com/checksum-property-for-autosar/.

[23] E. M. C. Jr., O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[24] J. Rushby, "An overview of formal verification for the time-triggered architecture," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, ser. LNCS, vol. 2469. Springer, Sep. 2002, pp. 83–105.

[25] A. Wick, "The HaLVM: A simple platform for simple platforms," Xen Summit Talk, August 2012, slides available at http://www-archive.xenproject.org/xensummit/xs12na_talks/M9b.html.

[26] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *SIGPLAN Notices*, vol. 48, no. 4, pp. 461–472, Mar. 2013.

[27] X. Zhang, O. Acimez, and J.-P. Seifert, "Building efficient integrity measurement and attestation for mobile phone platforms," in *Security and Privacy in Mobile Information and Communication Systems*, ser. LNCS. Springer, 2009, vol. 17, pp. 71–82.

[28] S. L. Kinney, *Trusted Platform Module Basics: Using TPM in Embedded Systems (Embedded Technology)*. Newnes, 2006.

[29] O. Bubeck and V. Bourgeois, "New security concepts for future generation automotive electronic control units," in *Proceedings of Embedded Real-time Software and Systems (ERTS) 2014, Toulouse, France*, 2014, available at http://www.erts2014.org/Site/0R4UXE94/Fichier/erts2014_7C2.pdf.

[30] L. Pike, A. Goodloe, R. Morisset, and S. Niller, "Copilot: A hard real-time runtime monitor," in *Proceedings of the 1st Intl. Conference on Runtime Verification*, ser. LNCS. Springer, November 2010.

[31] J. Rushby, "Bus architectures for safety-critical embedded systems," in *EMSOFT 2001: Proceedings of the First Workshop on Embedded Software*, ser. LNCS, vol. 2211. Springer, Oct. 2001, pp. 306–323.

[32] "NRSC-4-B: United States RBDS standard," April 2011, available at http://www.nrscstandards.org/SG/nrsc-4-B.pdf.

[33] "CVE-2013-3200," April 2013, available at http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-3200.

[34] A. Francillon, B. Danev, and S. Capkun, "Relay attacks on passive keyless entry and start systems in modern cars." in *NDSS*, 2011, available at http://s3.eurecom.fr/docs/ndss11_francillon.pdf.

[35] J. Harding, G. Powell, Y. R., F. R., D. J., S. C., L. D., S. M., J., and J. Wang, "Vehicle-to-vehicle communications: Readiness of V2V technology for application," U.S. Department of Transportation National Highway Traffic Safety Administration, August, Tech. Rep. DOT HS 812 014, 2014.

[36] A. Humayed and B. Luo, "Cyber-physical security for smart cars – issues, survey and challenges," in *2nd Intl. IFIP Workshop on Emerging Ideas and Trends in Engineering of Cyber-Physical Systems (EITEC)*, 2015.

[37] R. B. GmbH, "Can with flexible data-rate; version 1.1," Whitepaper, August 2011, available at http://www.bosch-semiconductors.de/media/pdf_1/canliteratur/can_fd.pdf.

[38] M. Dworkin, *Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC*. US Department of Commerce, National Institute of Standards and Technology, 2007.

[39] "Defense science board task force on high performance microchip supply," available at http://www.acq.osd.mil/dsb/reports/ADA435563.pdf.

[40] F. Kiamilev, "Demonstration of hardware trojans," http://www.defcon.org/images/defcon-16/dc16-presentations/defcon-16-kiamilev.pdf.

[41] Y. Shiyanovskii, F. G. Wolff, C. A. Papachristou, D. J. Weyer, and W. Clay, "Hardware trojan by hot carrier injection," *CoRR*, vol. abs/0906.3832, 2009, informal publication. [Online]. Available: http://arxiv.org/abs/0906.3832

[42] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and implementing malicious hardware," in *First USENIX Workshop on Large-Scale Exploits and Emergent Threats, April 15, 2008, San Francisco, CA, USA, Proceedings*, F. Monrose, Ed. USENIX Association, 2008. [Online]. Available: http://www.usenix.org/events/leet08/tech/full_papers/king/king.pdf

[43] S. Adee, "The hunt for the kill switch," *IEEE Spectrum*, vol. 45, no. 5, pp. 34–39, 2008.

[44] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design & Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MDT.2010.7

[45] S. Jha and S. K. Jha, "Randomization based probabilistic approach to detect trojan circuits," in *HASE*. IEEE Computer Society, 2008, pp. 117–124. [Online]. Available: http://dx.doi.org/10.1109/HASE.2008.37

[46] M. Potkonjak, A. Nahapetian, M. Nelson, and T. Massey, "Hardware trojan horse detection using gate-level characterization," in *DAC*. ACM, 2009, pp. 688–693. [Online]. Available: http://doi.acm.org/10.1145/1629911.1630091

[47] Y. Jin and Y. Makris, "Hardware trojan detection using path delay fingerprint," in *IEEE International Workshop on Hardware-Oriented Security and Trust, HOST 2008, Anaheim, CA, USA, June 9, 2008. Proceedings*, M. Tehranipoor and J. Plusquellic, Eds. IEEE Computer Society, 2008, pp. 51–57. [Online]. Available: http://dx.doi.org/10.1109/HST.2008.4559049

[48] S. Narasimhan, R. S. Chakraborty, D. Du, S. Paul, F. G. Wolff, C. A. Papachristou, K. Roy, and S. Bhunia, "Multiple-parameter side-channel analysis: A non-invasive hardware trojan detection approach," in *HOST 2010, Proceedings of the 2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), 13-14 June 2010, Anaheim Convention Center, California, USA*. IEEE Computer Society, 2010, pp. 13–18. [Online]. Available: http://dx.doi.org/10.1109/HST.2010.5513122

[49] F. G. Wolff, C. A. Papachristou, S. Bhunia, and R. S. Chakraborty, "Towards trojan-free trusted ICs: Problem analysis and detection scheme," in *DATE*. IEEE, 2008, pp. 1362–1365. [Online]. Available: http://dx.doi.org/10.1109/DATE.2008.4484928

[50] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *CoRR*, vol. abs/1409.7324, 2014. [Online]. Available: http://arxiv.org/abs/1409.7324