2010 IEEE Symposium on Security and Privacy

Experimental Security Analysis of a Modern Automobile

Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, and Tadayoshi Kohno
Department of Computer Science and Engineering
University of Washington
Seattle, Washington 98195–2350
Email: {supersat,aczeskis,franzi,shwetak,yoshi}@cs.washington.edu

Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage
Department of Computer Science and Engineering
University of California San Diego
La Jolla, California 92093–0404
Email: {s,dlmccoy,brian,d8anders,hovav,savage}@cs.ucsd.edu

***Abstract*—Modern automobiles are no longer mere mechanical devices; they are pervasively monitored and controlled by dozens of digital computers coordinated via internal vehicular networks. While this transformation has driven major advancements in efficiency and safety, it has also introduced a range of new potential risks. In this paper we experimentally evaluate these issues on a modern automobile and demonstrate the fragility of the underlying system structure. We demonstrate that an attacker who is able to infiltrate virtually any Electronic Control Unit (ECU) can leverage this ability to completely circumvent a broad array of safety-critical systems. Over a range of experiments, both in the lab and in road tests, we demonstrate the ability to adversarially control a wide range of automotive functions and completely ignore driver input — including disabling the brakes, selectively braking individual wheels on demand, stopping the engine, and so on. We find that it is possible to bypass rudimentary network security protections within the car, such as maliciously bridging between our car's two internal subnets. We also present composite attacks that leverage individual weaknesses, including an attack that embeds malicious code in a car's telematics unit and that will completely erase any evidence of its presence after a crash. Looking forward, we discuss the complex challenges in addressing these vulnerabilities while considering the existing automotive ecosystem.**

***Keywords*—Automobiles, communication standards, communication system security, computer security, data buses.**

I. Introduction

Through 80 years of mass-production, the passenger automobile has remained superficially static: a single gasoline-powered internal combustion engine; four wheels; and the familiar user interface of steering wheel, throttle, gearshift, and brake. However, in the past two decades the underlying control systems have changed dramatically. Today's automobile is no mere mechanical device, but contains a myriad of computers. These computers coordinate and monitor sensors, components, the driver, and the passengers. Indeed, one recent estimate suggests that the typical luxury sedan now contains over 100 MB of binary code spread across 50–70 independent computers — *Electronic Control Units* (*ECUs*) in automotive vernacular — in turn communicating over one or more shared *internal network buses* [8], [13].

While the automotive industry has always considered *safety* a critical engineering concern (indeed, much of this new software has been introduced specifically to *increase* safety, e.g., *Anti-lock Brake Systems*) it is not clear whether vehicle manufacturers have anticipated in their designs the possibility of an adversary. Indeed, it seems likely that this increasing degree of computerized control also brings with it a corresponding array of potential threats.

Compounding this issue, the attack surface for modern automobiles is growing swiftly as more sophisticated services and communications features are incorporated into vehicles. In the United States, the federally-mandated *On-Board Diagnostics* (*OBD-II*) port, under the dash in virtually all modern vehicles, provides direct and standard access to internal automotive networks. User-upgradable subsystems such as audio players are routinely attached to these same internal networks, as are a variety of short-range wireless devices (Bluetooth, wireless tire pressure sensors, etc.). Telematics systems, exemplified by General Motors' (GM's) OnStar, provide value-added features such as automatic crash response, remote diagnostics, and stolen vehicle recovery over a long-range wireless link. To do so, these telematics systems integrate internal automotive subsystems with a remote command center via a wide-area cellular connection. Some have taken this concept even further — proposing a "car as a platform" model for third-party development. Hughes Telematics has described plans for developing an "App Store" for automotive applications [22] while Ford recently announced that it will open its Sync telematics system as a platform for third-party applications [14]. Finally, proposed future vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2X) communications systems [5], [6], [7], [25] will only broaden the attack surface further.

1081-6011/10 $26.00 © 2010 IEEE
DOI 10.1109/SP.2010.34

Overall, these trends suggest that a wide range of vectors will be available by which an attacker might compromise a component and gain access to internal vehicular networks — with unknown consequences. Unfortunately, while previous research efforts have largely considered vehicular security risks in the abstract, very little is publicly known about the practical security issues in automobiles on the road today. Our research aims to fill this gap.

This paper investigates these issues through an empirical lens — with active experiments against two late-model passenger cars (same make and model). We test these cars' components in isolation in the lab, as a complete system in a controlled setting (with the car elevated on jacks), and in live road tests on a closed course. We have endeavored to comprehensively assess how much resilience a conventional automobile has against a digital attack mounted against its internal components. Our findings suggest that, unfortunately, the answer is "little."

Indeed, we have demonstrated the ability to systematically control a wide array of components including engine, brakes, heating and cooling, lights, instrument panel, radio, locks, and so on. Combining these we have been able to mount attacks that represent potentially significant threats to personal safety. For example, we are able to forcibly and completely disengage the brakes while driving, making it difficult for the driver to stop. Conversely, we are able to forcibly activate the brakes, lurching the driver forward and causing the car to stop suddenly.

Rather than focus just on individual attacks, we conduct a comprehensive analysis of our cars' digital components and internal networks. We experimentally evaluate the security properties of each of the key components within our cars, and we analyze the security properties of the underlying network substrate. Beyond measuring the real threats against the computerized components within modern cars, as well as the fundamental reasons those threats are possible, we explore considerations and directions for reconciling the tension between strategies for better security and the broader context surrounding automobiles.

II. Background

There are over 250 million registered passenger automobiles in the United States [4]. The vast majority of these are computer controlled to a significant degree and virtually all new cars are now pervasively computerized. However, in spite of their prevalence, the structure of these systems, the functionality they provide and the networks they use internally are largely unfamiliar to the computer security community. In this section, we provide basic background context concerning automotive embedded systems architecture in general and an overview of prior related work concerning automotive security.

A. Automotive Embedded Systems

Digital control, in the form of self-contained embedded systems called *Engine Control Units* (*ECUs*), entered US production vehicles in the late 1970s, largely due to requirements of the California Clean Air Act (and subsequent federal legislation) and pressure from increasing gasoline prices [21]. By dynamically measuring the oxygen present in exhaust fumes, the ECU could then adjust the fuel/oxygen mixture before combustion, thereby improving efficiency and reducing pollutants. Since then, such systems have been integrated into virtually every aspect of a car's functioning and diagnostics, including the throttle, transmission, brakes, passenger climate and lighting controls, external lights, entertainment, and so on, causing the term ECU to be generalized to *Electronic Control Units*. Thus, over the last few decades the amount of software in luxury sedans has grown from virtually nothing to tens of millions of lines of code, spread across 50–70 independent ECUs [8].

ECU Coupling. Many features require complex interactions *across* ECUs. For example, modern *Electronic Stability Control* (*ESC*) systems monitor individual wheel speed, steering angle, throttle position, and various accelerometers. The ESC automatically modulates engine torque and wheel speed to increase traction when the car's line stops following the steering angle (i.e., a skid). If brakes are applied they must also interact with the *Anti-lock Braking System* (*ABS*). More advanced versions also offer *Roll Stability Control* (*RSC*), which may also apply brakes, reduce the throttle, and modulate the steering angle to prevent the car from rolling over. *Active Cruise Control* (*ACC*) systems scan the road ahead and automatically increase or decrease the throttle (about some pre-programmed cruising speed) depending on the presence of slower vehicles in the path (e.g., the Audi Q7 will automatically apply brakes, completely stopping the vehicle if necessary, with no user input). Versions of this technology also provide "pre-crash" features in some cars including pre-charging brakes and pre-tensioning seat belts. Some new luxury sedans (e.g., the Lexus LS460) even offer automated parallel parking features in which steering is completely subsumed. These trends are further accelerated by electric-driven vehicles that require precise software control over power management and regenerative braking to achieve high efficiency, by a slew of emerging safety features, such as VW's Lane Assist system, and by a wide range of proposed entertainment and communications features (e.g., it was recently announced that GM's OnStar will offer integration with Twitter [10]). Even full "steer-by-wire" functionality has been seen in a range of concept cars including GM's widely publicized Hywire fuel cell vehicle [12].

While some early systems used one-off designs and bilateral physical wire connections for such interactions (e.g., between different sensors and an ECU), this approach

does not scale. A combination of time-to-market pressures, wiring overhead, interaction complexity, and economy of scale pressures have driven manufacturers and suppliers to standardize on a few key digital buses, such as *Controller Area Network* (*CAN*) and FlexRay, and software technology platforms (cf. Autosar [1]) shared across component manufacturers and vendors. Indeed, the distributed nature of the automotive manufacturing sector has effectively mandated such an approach — few manufacturers can afford the overhead of full soup-to-nuts designs anymore.

Thus, the typical car contains multiple buses (generally based on the CAN standard) covering different component groups (e.g., a high-speed bus may interconnect powertrain components that generate real-time telemetry while a separate low-speed bus might control binary actuators like lights and doors). While it seems that such buses could be physically isolated (e.g., safety critical systems on one, entertainment on the other), in practice they are "bridged" to support subtle interaction requirements. For example, consider a car's *Central Locking Systems* (*CLS*), which controls the power door locking mechanism. Clearly this system must monitor the physical door lock switches, wireless input from any remote key fob (for keyless entry), and remote telematics commands to open the doors. However, unintuitively, the CLS must *also* be interconnected with safety critical systems such as crash detection to ensure that car locks are disengaged after airbags are deployed to facilitate exit or rescue.

Telematics. Starting in the mid-1990's automotive manufacturers started marrying more powerful ECUs — providing full Unix-like environments — with peripherals such as *Global Positioning Systems* (*GPS*), and adding a "reach-back" component using cellular back-haul links. By far the best known and most innovative of such systems is GM's OnStar, which — now in its 8th generation — provides a myriad of services. An OnStar-equipped car can, for example, analyze the car's *On Board Diagnostics* (*OBD*) as it is being driven, proactively detect likely vehicle problems, and notify the driver that a trip to the repair shop is warranted. OnStar ECUs monitor crash sensors and will automatically place emergency calls, provide audio-links between passengers and emergency personnel, and relay GPS-based locations. These systems even enable properly authorized OnStar personnel to remotely unlock cars, track the cars' locations and, starting with some 2009 model years, remotely stop them (for the purposes of recovery in case of theft) purportedly by stopping the flow of fuel to the engines. To perform these functions, OnStar units routinely bridge all important buses in the automobile, thereby maximizing flexibility, and implement an on-demand link to the Internet via Verizon's digital cellular service. However, GM is by no means unique and virtually every manufacturer now has a significant telematics package in their lineup (e.g., Ford's Sync, Chrysler's UConnect, BMW's Connected Drive, and Lexus's Enform), frequently provided in collaboration with third-party specialist vendors such as Hughes Telematics and ATX Group.

Taken together, ubiquitous computer control, distributed internal connectivity, and telematics interfaces increasingly combine to provide an application software platform with external network access. There are thus ample reasons to reconsider the state of vehicular computer security.

B. Related Work

Indeed, we are not the first to observe the potential fragility of the automotive environment. In the academic context, several groups have described potential vulnerabilities in automotive systems, e.g., [19], [24], [26], [27], [28]. They provide valuable contributions toward framing the vehicle security and privacy problem space — notably in outlining the security limitations of the popular CAN bus protocol — as well as possible directions for securing vehicle components. With some exceptions, e.g., [15], most of these efforts consider threats abstractly; considering "what-if" questions about a hypothetical attacker. Part of our paper's contribution is to make this framing concrete by providing comprehensive experimental results assessing the behavior of real automobiles and automotive components in response to specific attacks.

Further afield, a broad array of researchers have considered the security problems of vehicle-to-vehicle (V2V) systems (sometimes also called vehicular ad-hoc networks, or VANETs); see [18] for a survey. Indeed, this work is critical, as such future networks will otherwise present yet another entry point by which attackers might infiltrate a vehicle. However, our work is focused squarely on the possibilities *after* any such infiltration. That is, what are the security issues *within* a car, rather than external to it.

Still others have focused on theft-related access control mechanisms, including successful attacks against vehicle keyless entry systems [11], [16] and vehicle immobilizers [3].

Outside the academic realm, there is a small but vibrant "tuner" subculture of automobile enthusiasts who employ specialized software to improve performance (e.g., by removing electronic RPM limitations or changing spark timings, fuel ignition parameters, or valve timings) frequently at the expense of regulatory compliance [20], [23]. These groups are not adversaries; their modifications are done to improve and personalize their own cars, not to cause harm. In our work, we consider how an adversary with malicious motives might disrupt or modify automotive systems.

Finally, we point out that while there is an emerging effort focused on designing fully autonomous vehicles (e.g., DARPA Grand Challenge [9]), these are specifically

designed to be robotically controlled. While such vehicles would undoubtedly introduce yet new security concerns, in this paper we concern ourselves solely with the vulnerabilities in today's commercially-available automobiles.

C. Threat Model

In this paper we intentionally and explicitly skirt the question of a "threat model." Instead, we focus primarily on what an attacker could do to a car *if* she was able to maliciously communicate on the car's internal network. That said, this does beg the question of *how* she might be able to gain such access.

While we leave a full analysis of the modern automobile's attack surface to future research, we briefly describe here the two "kinds" of vectors by which one might gain access to a car's internal networks.

The first is physical access. Someone — such as a mechanic, a valet, a person who rents a car, an ex-friend, a disgruntled family member, or the car owner — can, with even momentary access to the vehicle, insert a malicious component into a car's internal network via the ubiquitous OBD-II port (typically under the dash). The attacker may leave the malicious component permanently attached to the car's internal network or, as we show in this paper, they may use a brief period of connectivity to embed the malware within the car's existing components and then disconnect. A similar entry point is presented by counterfeit or malicious components entering the vehicle parts supply chain — either before the vehicle is sent to the dealer, or with a car owner's purchase of an aftermarket third-party component (such as a counterfeit FM radio).

The other vector is via the numerous wireless interfaces implemented in the modern automobile. In our car we identified no fewer than five kinds of digital radio interfaces accepting outside input, some over only a short range and others over indefinite distance. While outside the scope of this paper, we wish to be clear that vulnerabilities in such services are not purely theoretical. We have developed the ability to remotely compromise key ECUs in our car via externally-facing vulnerabilities, amplify the impact of these remote compromises using the results in this paper, and ultimately monitor and control our car remotely over the Internet.

III. Experimental Environment

Our experimental analyses focus on two 2009 automobiles of the same make and model.[1] We selected our particular vehicle because it contained both a large number of electronically-controlled components (necessitated by complex safety features such as anti-lock brakes and stability control) and a sophisticated telematics system. We purchased two vehicles to allow differential testing and to validate that our results were not tied to one individual vehicle. At times we also purchased individual replacement ECUs via third-party dealers to allow additional testing. Table I lists some of the most important ECUs in our car.

We experimented with these cars — and their internal components — in three principal settings:

- *Bench.* We physically extracted hardware from the car for analysis in our lab. As with most automobile manufacturers, our vehicles use a variant of the Controller Area Network (CAN) protocol for communicating among vehicle components (in our case both a high-speed and low-speed variant as well as a variety of proprietary higher-layer network management services). Through this protocol, any component can be accessed and interrogated in isolation in the lab. Figure 1 shows an example setup, with the Electronic Brake Control Module (EBCM) hooked up to a power supply, a CAN-to-USB converter, and an oscilloscope.
- *Stationary car.* We conducted most of our in-car experiments with the car stationary. For both safety and convenience, we elevated the car on jack stands for experiments that required the car to be "at speed"; see Figure 3.

  Figure 2 shows the experimental setup inside the car. For these experiments, we connected a laptop to the car's standard On-Board Diagnostics II (OBD-II) port. We used an off-the-shelf CAN-to-USB interface (the CANCapture ECOM cable) to interact with the car's high-speed CAN network, and an Atmel AT90CAN128 development board (the Olimex AVR-CAN) with custom firmware to interact with the car's low-speed CAN network. The laptop ran our custom CarShark program (see below).
- *On the road.* To obtain full experimental fidelity, for some of our results we experimented at speed while on a closed course.

  We exercised numerous precautions to protect the safety of both our car's driver and any third parties. For example, we used the runway of a de-commissioned airport because the runway is long and straight, giving us additional time to respond should an emergency situation arise (see Figure 7).

  For these experiments, one of us drove the car while three others drove a chase car on a parallel service road; one person drove the chase car, one documented much of the process on video, and one wirelessly controlled the test car via an 802.11 *ad hoc* connection to a laptop in the test car that in turn accessed its CAN bus.

[1] We believe the risks identified in this paper arise from the *architecture* of the modern automobile and not simply from design decisions made by any single manufacturer. For this reason, we have chosen not to identify the particular make and model used in our tests. We believe that other automobile manufacturers and models with similar features may have similar security properties.

Component	Functionality	Low-Speed Comm. Bus	High-Speed Comm. Bus
ECM	*Engine Control Module* Controls the engine using information from sensors to determine the amount of fuel, ignition timing, and other engine parameters.		✓
EBCM	*Electronic Brake Control Module* Controls the Antilock Brake System (ABS) pump motor and valves, preventing brakes from locking up and skidding by regulating hydraulic pressure.		✓
TCM	*Transmission Control Module* Controls electronic transmission using data from sensors and from the ECM to determine when and how to change gears.		✓
BCM	*Body Control Module* Controls various vehicle functions, provides information to occupants, and acts as a firewall between the two subnets.	✓	✓
Telematics	*Telematics Module* Enables remote data communication with the vehicle via cellular link.	✓	✓
RCDLR	*Remote Control Door Lock Receiver* Receives the signal from the car's key fob to lock/unlock the doors and the trunk. It also receives data wirelessly from the Tire Pressure Monitoring System sensors.	✓	
HVAC	*Heating, Ventilation, Air Conditioning* Controls cabin environment.	✓	
SDM	*Inflatable Restraint Sensing and Diagnostic Module* Controls airbags and seat belt pretensioners.	✓	
IPC/DIC	*Instrument Panel Cluster/Driver Information Center* Displays information to the driver about speed, fuel level, and various alerts about the car's status.	✓	
Radio	*Radio* In addition to regular radio functions, funnels and generates most of the in-cabin sounds (beeps, buzzes, chimes).	✓	
TDM	*Theft Deterrent Module* Prevents vehicle from starting without a legitimate key.	✓	

Table I. Key Electronic Control Units (ECUs) within our cars, their roles, and which CAN buses they are on.

The CARSHARK *Tool.* To facilitate our experimental analysis, we wrote CARSHARK — a custom CAN bus analyzer and packet injection tool (see Figure 4). While there exist commercially available CAN sniffers, none were appropriate for our use. First, we needed the ability to process and manipulate our vendor's proprietary extensions to the CAN protocol. Second, while we could have performed limited testing using a commercial CAN sniffer coupled with a manufacturer-specific diagnostic service tool, this combination still doesn't offer the flexibility to support our full range of attack explorations, including reading out ECU memory, loading custom code into ECUs, or generating fuzz-testing packets over the CAN interface.

IV. INTRA-VEHICLE NETWORK SECURITY

Before experimentally evaluating the security of individual car components, we assess the security properties of the CAN bus in general, which we describe below. We do so by first considering weaknesses inherent to the protocol stack and then evaluating the degree to which our car's components comply with the standard's specifications.

A. CAN Bus

There are a variety of protocols that can be implemented on the vehicle bus, but starting in 2008 all cars sold in the U.S. are required to implement the Controller Area Network (CAN) bus (ISO 11898 [17]) for diagnostics. As a result, CAN — roughly speaking, a link-layer data protocol — has become the dominant communication network for in-car networks (e.g., used by BMW, Ford, GM, Honda, and Volkswagen).

A CAN packet (shown in Figure 5) does not include addresses in the traditional sense and instead supports a publish-and-subscribe communications model. The *CAN ID* header is used to indicate the packet type, and each packet is both physically and logically broadcast to all nodes, which then decide for themselves whether to process the packets.

The CAN variant for our car includes slight extensions to framing (e.g., on the interpretation of certain CAN ID's) and two separate physical layers — a *high-speed* bus which is differentially-signaled and primarily used by powertrain systems and a *low-speed* bus (SAE J2411) using a single wire and supporting less-demanding components. When necessary, a gateway bridge can route selected data between

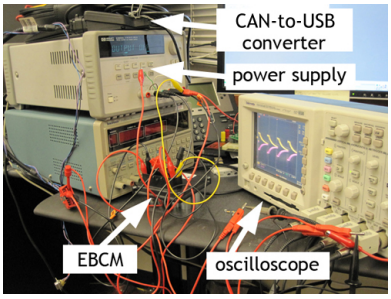

Figure 1. Example bench setup within our lab. The Electronic Brake Control Module (ECBM) is hooked up to a power supply, a CAN-to-USB converter, and an oscilloscope.

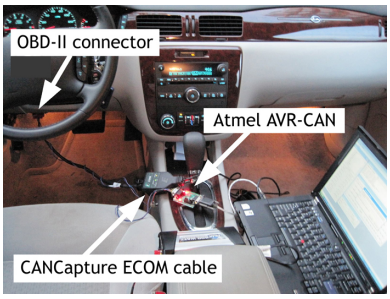

Figure 2. Example experimental setup. The laptop is running our custom CARSHARK CAN network analyzer and attack tool. The laptop is connected to the car's OBD-II port.

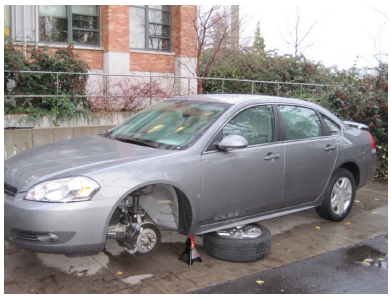

Figure 3. To test ECU behavior in a controlled environment, we immobilized the car on jack stands while mounting attacks.

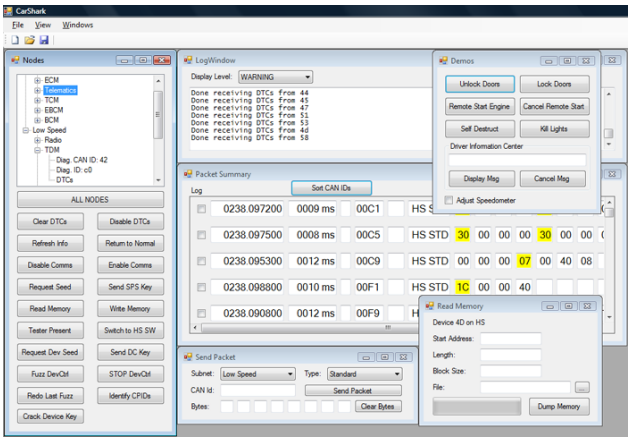

Figure 4. Screenshot of the CARSHARK interface. CARSHARK is being used to sniff the CAN bus. Values that have been recently updated are in yellow. The left panel lists all recognized nodes on high and low speed subnets of the CAN bus and has some action buttons. The demo panel on the right provides some proof-of-concept demos.

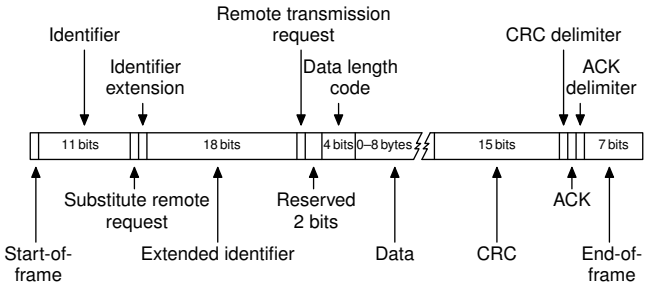

Figure 5. CAN packet structure. Extended frame format is shown. Base frame format is similar.

the two buses. Finally, the protocol standards define a range of services to be implemented by ECUs.

B. CAN Security Challenges

The underlying CAN protocol has a number of inherent weaknesses that are common to any implementation. Key among these:

Broadcast Nature. Since CAN packets are both physically and logically broadcast to all nodes, a malicious component on the network can easily snoop on all communications or send packets to any other node on the network. CARSHARK leverages this property, allowing us to observe and reverse-engineer packets, as well as to inject new packets to induce various actions.

Fragility to DoS. The CAN protocol is extremely vulnerable to denial-of-service attacks. In addition to simple packet flooding attacks, CAN's priority-based arbitration scheme allows a node to assert a "dominant" state on the bus indefinitely and cause all other CAN nodes to back off. While most controllers have logic to avoid accidentally breaking the network this way, adversarially-controlled hardware would not need to exercise such precautions.

No Authenticator Fields. CAN packets contain no authenticator fields — or even any source identifier fields — meaning that any component can indistinguishably send a packet to any other component. This means that any single compromised component can be used to control all of the other components on that bus, provided those components themselves do not implement defenses; we consider the security of individual components in Section V.

Weak Access Control. The protocol standards for our car specify a challenge-response sequence to protect ECUs against certain actions without authorization. A given ECU may participate in zero, one, or two challenge-response pairs:

- *Reflashing and memory protection.* One challenge-response pair restricts access to reflashing the ECU and reading out sensitive memory. By design, a service shop might authenticate with this challenge-response pair in order to upgrade the firmware on an ECU.
- *Tester capabilities.* Modern automobiles are complex and thus diagnosing their problems requires significant support. Thus, a major use of the CAN bus is in providing diagnostic access to service technicians. In particular, external test equipment (the "tester") must be able to interrogate the internal state of the car's components and, at times, manipulate this state as well.

Our car implements this capability via the *DeviceControl* service which is accessed in an RPC-like fashion directly via CAN messages. In our car, the second challenge-response pair described above is designed to restrict access to the DeviceControl services.

Under the hood, ECUs are supposed to use a fixed challenge (seed) for each of these challenge-response pairs; the corresponding responses (keys) are also fixed and stored in these ECUs. The motivation for using fixed seeds and keys is to avoid storing the challenge-response algorithm in the ECU firmware itself (since that firmware could be read out if an external flash chip is used). Indeed, the associated reference standard states "under no circumstances shall the encryption algorithm ever reside in the node." (The tester, however, does have the algorithm and uses it to compute the key.) Different ECUs should have different seeds and keys.

Despite these apparent security precautions, to the best of our knowledge many of the seed-to-key algorithms in use today are known by the car tuning community.

Furthermore, as described in the protocol standards, the challenges (seeds) and responses (keys) are both just 16 bits. Because the ECUs are required to allow a key attempt every 10 seconds, an attacker could crack one ECU key in a little over seven and a half days. If an attacker has access to the car's network for this amount of time (such as through another compromised component), any reflashable ECU can be compromised. Multiple ECUs can be cracked in parallel, so this is an upper bound on the amount of time it could take to crack a key in *every* ECU in the vehicle. Furthermore, if an attacker can physically remove a component from the car, she can further reduce the time needed to crack a component's key to roughly three and a half days by powercycling the component every two key attempts (we used this approach to perform an exhaustive search for the Electronic Brake Control Module (EBCM) key on one of our cars, recovering the key in about a day and a half; see Figure 1 for our experimental setup).

In effect, there are numerous realistic scenarios in which the challenge-response sequences defined in the protocol specification can be circumvented by a determined attacker.

ECU Firmware Updates and Open Diagnostic Control. Given the generic weaknesses with the aforementioned access control mechanisms, it is worth stepping back and reconsidering the benefits and risks associated with exposing ECUs to reflashing and diagnostic testing.

First, the ability to do software-only upgrades to ECUs can be extremely valuable to vehicle manufacturers, who might otherwise have to bear the cost of physically replacing ECUs for trivial defects in the software. For example, one of us recently received a letter from a car dealer, inviting us to visit an auto shop in order to upgrade the firmware on our personal car's ECM to correctly meet certain emission requirements. However, it is also well known that attackers can use software updates to inject malicious code into systems [2]. The challenge-response sequences alone are not sufficient to protect against malicious firmware updates; in subsequent sections we investigate whether additional protection mechanisms are deployed at a higher level (such as the cryptographically signed firmware updates).

Similarly, the DeviceControl service is a tremendously powerful tool for assisting in the diagnosis of a car's components. But, given the generic weaknesses of the CAN access control mechanisms, the DeviceControl capabilities present enumerable opportunities to an attacker (indeed, a great number of our attacks are built on DeviceControl). In many ways this challenge parallels the security vs. functionality tension presented by debuggers in conventional operating systems; to be effective debuggers need to be able to examine and manipulate all state, but if they can do that they can do anything. However, while traditional operating systems generally finesse this problem via access-control rights on a per-user basis, there is no equivalent concept in CAN. Given the weaknesses with the CAN access control sequence, the role of "tester" is effectively open to any node on the bus and thus to any attacker.

Worse, in Section IV-C below we find that many ECUs in our car deviate from their own protocol standards, making it even easier for an attacker to initiate firmware updates or DeviceControl sequences — without even needing to bypass the challenge-response protocols.

C. Deviations from Standards

In several cases, our car's protocol standards do prescribe risk-mitigation strategies with which components should comply. However, our experimental findings revealed that not all components in the car always follow these specifications.

Disabling Communications. For example, the standard states that ECUs should reject the "disable CAN communications" command when it is unsafe to accept and act on it, such as when a car is moving. However, we experimentally verified that this is not actually the case in our car: we were able to disable communications to and from all the ECUs in Table I even with the car's wheels moving at speed on jack stands and while driving on the closed road course.

Reflashing ECUs While Driving. The standard also states that ECUs should reject reflashing events if they deem them unsafe. In fact, it states: "The engine control module should reject a request to initiate a programming event if the engine were running." However, we experimentally verified that we could place the Engine Control Module (ECM) and Transmission Control Module (TCM) into reflashing mode when our car was at speed on jack stands. When the ECM enters this mode, the engine stops running. We also verified that we could place the ECM into reflashing mode while driving on the closed course.

Noncompliant Access Control: Firmware and Memory. The standard states that ECUs with emissions, anti-theft, or safety functionality *must* be protected by a challenge-response access control protocol (as per Section IV-B).

Even disregarding the weakness of this protocol, we found it was implemented less broadly than we would have expected. For example, the telematics unit in our car, which are connected to the car's CAN buses, use a hardcoded challenge and a hardcoded response common to *all* similar units, seemingly in violation of the standard (specifically, the standard states that "all nodes with the same part number shall NOT have the same security seed"). Even worse, the result of the challenge-response protocol is never used anywhere; one can reflash the unit at any time without completing the challenge-response protocol. We verified experimentally that we can load our own code onto our car's telematics unit *without* authenticating.

Some access-controlled operations, such as reading sensitive memory areas (such as the ECU's program or keys) may be outright denied if deemed too risky. For example, the standard states that an ECU can define memory addresses that "[it] will not allow a tester to read under any circumstances (e.g., the addresses that contain the security seed and key values)." However, in another instance of non-compliance, we experimentally verified that we could read the reflashing keys out of the BCM without authenticating, and the DeviceControl keys for the ECM and TCM just by authenticating with the reflashing key. We were also able to extract the telematics units' entire memory, including their keys, without authentication.

Noncompliant Access Control: Device Overrides. Recall that the DeviceControl service is used to override the state of components. However, ECUs are expected to reject unsafe DeviceControl override requests, such as releasing the brakes when the car is in motion (an example mentioned in the standard). Some of these unsafe overrides are needed for testing during the manufacturing process, so those can be enabled by authenticating with the DeviceControl key. However, we found during our experiments that certain unsafe device control operations succeeded without authenticating; we summarize these in Tables II, V-A, and IV.

Imperfect Network Segregation. The standard implicitly defines the high-speed network as more trusted than the low-speed network. This difference is likely due to the fact that the high-speed network includes the real-time safety-critical components (e.g., engine, brakes), while the low-speed network commonly includes components less critical to safety, like the radio and the HVAC system.

The standard states that gateways between the two networks must only be re-programmable from the high-speed network, presumably to prevent a low-speed device from compromising a gateway to attack the high-speed network. In our car, there are two ECUs which are on both buses and can potentially bridge signals: the Body Controller Module (BCM) and the telematics unit. While the telematics unit is not technically a gateway, it connects to both networks and can only be reprogrammed (against the spirit of the standard) from the low-speed network, allowing a low-speed device to attack the high-speed network through the telematics unit. We verified that we could bridge these networks by uploading code to the telematics unit from the low-speed network that, in turn, sent packets on the high-speed network.

V. Component Security

We now examine individual components on our car's CAN network, and what an attacker could do by communicating with each one individually. We discuss compound attacks involving multiple components in Section VI. We omit certain details (such as complete packet payloads) to prevent would-be attackers from using our results directly.

A. Attack Methodology

Recall that Table I gives an overview of our car's critical components, their functionality, and whether they are on the car's high-speed or low-speed CAN subnet. For each of these components, our methodology for formulating attacks consisted of some or all of the following three major approaches, summarized below.

Packet Sniffing and Targeted Probing. To begin, we used CarShark to observe traffic on the CAN buses in order to determine how ECUs communicate with each other. This also revealed to us which packets were sent as we activated various components (such as turning on the headlights). Through a combination of replay and informed probing, we were able to discover how to control the radio, the Instrument Panel Cluster (IPC), and a number of the Body Control Module (BCM) functions, as we discuss below. This approach worked well for packets that come up during normal operation, but was less useful in mapping the interface to safety-critical powertrain components.

Fuzzing. Much to our surprise, significant attacks do not require a complete understanding or reverse-engineering of even a single component of the car. In fact, because the range of valid CAN packets is rather small, significant damage can be done by simple fuzzing of packets (i.e., iterative testing of random or partially random packets). Indeed, for attackers seeking indiscriminate disruption, fuzzing is an effective attack by itself. (Unlike traditional uses of fuzzing, we use fuzzing to aid in the reverse engineering of functionality.)

As mentioned previously, the protocol standards for our car define a CAN-based service called *DeviceControl*, which allows testing devices (used during manufacturing quality control or by mechanics) to override the normal output functionality of an ECU or reset some learned internal

state. The DeviceControl service takes an argument called a *Control Packet Identifier* (CPID), which specifies a group of controls to override. Each CPID can take up to five bytes as parameters, specifying which controls in the group are being overridden, and how to override them. For example, the Body Control Module (BCM) exports controls for the various external lights (headlights, brakelights, etc.) and their associated brightness can be set via the parameter data.

We discovered many of the DeviceControl functions for select ECUs (specifically, those controlling the engine (ECM), body components (BCM), brakes (EBCM), and heating and air conditioning (HVAC) systems) largely by fuzz testing. After enumerating all supported CPIDs for each ECU, we sent random data as an argument to valid CPIDs and correlated input bits with behaviors.

Reverse-Engineering. For a small subset of ECUs (notably the telematics unit, for which we obtained multiple instances via Internet-based used parts resellers) we dumped their code via the CAN *ReadMemory* service and used a third-party debugger (IDA Pro) to explicitly understand how certain hardware features were controlled. This approach is essential for attacks that require *new* functionality to be added (e.g., bridging low and high-speed buses) rather than simply manipulating existing software capabilities.

B. Stationary Testing

We now describe the results of our experiments with controlling critical components of the car. All initial experiments were done with the car stationary, in many cases immobilized on jack stands for safety, as shown in Figure 3. Some of our results are summarized in Tables II, V-A, and IV for fuzzing, and in Table V for other results. Tables II, V-A, and IV indicate the packet that was sent to the corresponding module, the resulting action, and four additional pieces of information: (1) Can the result of this packet be overridden manually, such as by pulling the physical door unlock knob, pushing on the brakes, or some other action? A *No* in this column means that we have found no way to manually override the result. (2) Does this packet have the same effect when the car is at speed? For this column, "at speed" means when the car was up on jack stands but the throttle was applied to bring the wheel speed to 40 MPH. (3) Does the module in question need to be unlocked with its DeviceControl key before these packets can elicit results? The fourth (4) additional column reflects our experiments during a live road test, which we will turn to in subsection V-C. Table V is similar, except that only the Kill Engine result is caused by a DeviceControl packet; we did not need to unlock the ECU before initiating this DeviceControl packet.

All of the controlled experiments were initially conducted on one car, and then all were repeated on our second car (road tests were only performed with the first car).

Figure 6. Displaying an arbitrary message and a false speedometer reading on the Driver Information Center. Note that the car is in Park.

Radio. One of the first attacks we discovered was how to control the radio and its display. We were able to completely control — and disable user control of — the radio, and to display arbitrary messages. For example, we were able to consistently increase the volume and prevent the user from resetting it. As the radio is also the component which controls various car sounds (e.g., turn signal clicks and seat belt warning chimes), we were also able to produce clicks and chimes at arbitrary frequencies, for various durations, and at different intervals. Table V presents some of these results.

Instrument Panel Cluster. We were able to fully control the Instrument Panel Cluster (IPC). We were able to display arbitrary messages, falsify the fuel level and the speedometer reading, adjust the illumination of instruments, and so on (also shown in Table V). For example, Figure 6 shows the instrument panel display with a message that we set by sending the appropriate packets over the CAN network. We discuss a more sophisticated attack using our control over the speedometer in Section VI.

Body Controller. Control of the BCM's function is split across the low-speed and high-speed buses. By reverse-engineering packets sent on the low-speed bus (Table V) and by fuzzing packets on the high-speed bus (as summarized in Table II), we were able to control essentially all of the BCM's functions. This means that we were able to discover packets to: lock and unlock the doors; jam the door locks by continually activating the lock relay; pop the trunk; adjust interior and exterior lighting levels; honk the horn (indefinitely and at varying frequencies); disable and enable the window relays; disable and enable the windshield wipers; continuously shoot windshield fluid; and disable the key lock relay to lock the key in the ignition.

Engine. Most of the attacks against the engine were found by fuzzing DeviceControl requests to the ECM. These findings are summarized in Table V-A. We were able to boost the engine RPM temporarily, disturb engine timing by resetting the learned crankshaft angle sensor error, disable

Packet	Result	Manual Override	At Speed	Need to Unlock	Tested on Runway
07 AE ... 1F 87	Continuously Activates Lock Relay	Yes	Yes	No	✓
07 AE ... C1 A8	Windshield Wipers On Continuously	No	Yes	No	✓
07 AE ... 77 09	Pops Trunk	No	Yes	No	✓
07 AE ... 80 1B	Releases Shift Lock Solenoid	No	Yes	No	
07 AE ... D8 7D	Unlocks All Doors	Yes	Yes	No	
07 AE ... 9A F2	Permanently Activates Horn	No	Yes	No	✓
07 AE ... CE 26	Disables Headlights in Auto Light Control	Yes	Yes	No	✓
07 AE ... 34 5F	All Auxiliary Lights Off	No	Yes	No	
07 AE ... F9 46	Disables Window and Key Lock Relays	No	Yes	No	
07 AE ... F8 2C	Windshield Fluid Shoots Continuously	No	Yes	No	✓
07 AE ... 15 A2	Controls Horn Frequency	No	Yes	No	
07 AE ... 15 A2	Controls Dome Light Brightness	No	Yes	No	
07 AE ... 22 7A	Controls Instrument Brightness	No	Yes	No	
07 AE ... 00 00	All Brake/Auxiliary Lights Off	No	Yes	No	✓
07 AE ... 1D 1D	Forces Wipers Off and Shoots Windshield Fluid Continuously	Yes†	Yes	No	✓

Table II. Body Control Module (BCM) DeviceControl Packet Analysis. This table shows BCM DeviceControl packets and their effects that we discovered during fuzz testing with one of our cars on jack stands. A ✓in the last column indicates that we also tested the corresponding packet with the driving on a runway. A "Yes" or "No" in the columns "Manual Override," "At Speed," and "Need to Unlock" indicate whether or not (1) the results could be manually overridden by a car occupant, (2) the same effect was observed with the car at speed (the wheels spinning at about 40 MPH and/or on the runway), and (3) the BCM needed to be unlocked with its DeviceControl key.
†The highest setting for the windshield wipers cannot be disabled and serves as a manual override.

Packet	Result	Manual Override	At Speed	Need to Unlock	Tested on Runway
07 AE ... E5 EA	Initiate Crankshaft Re-learn; Disturb Timing	Yes	Yes	Yes	
07 AE ... CE 32	Temporary RPM Increase	No	Yes	Yes	✓
07 AE ... 5E BD	Disable Cylinders, Power Steering/Brakes	Yes	Yes	Yes	
07 AE ... 95 DC	Kill Engine, Cause Knocking on Restart	Yes	Yes	Yes	✓
07 AE ... 8D C8	Grind Starter	No	Yes	Yes	
07 AE ... 00 00	Increase Idle RPM	No	Yes	Yes	✓

Table III. Engine Control Module (ECM) DeviceControl Packet Analysis. This table is similar to Table II.

Packet	Result	Manual Override	At Speed	Need to Unlock†	Tested on Runway
07 AE ... 25 2B	Engages Front Left Brake	No	Yes	Yes	✓
07 AE ... 20 88	Engages Front Right Brake/Unlocks Front Left	No	Yes	Yes	✓
07 AE ... 86 07	Unevenly Engages Right Brakes	No	Yes	Yes	✓
07 AE ... FF FF	Releases Brakes, Prevents Braking	No	Yes	Yes	✓

Table IV. Electronic Brake Control Module (EBCM) DeviceControl Packet Analysis. This table is similar to Table II.
†The EBCM did not need to be unlocked with its DeviceControl key when the car was on jack stands. Later, when we tested these packets on the runway, we discovered that the EBCM rejected these commands when the speed of the car exceeded 5 MPH without being unlocked.

Destination ECU	Packet	Result	Manual Override	At Speed	Tested on Runway
IPC	00 00 ... 00 00	Falsify Speedometer Reading	No	Yes	✓
Radio	04 00 ... 00 00	Increase Radio Volume	No	Yes	
Radio	63 01 ... 39 00	Change Radio Display	No	Yes	
IPC	00 02 ... 00 00 27 01 ... 65 00	Change DIC Display	No	Yes	
BCM	04 03	Unlock Car†	Yes	Yes	
BCM	04 01	Lock Car†	Yes	Yes	
BCM	04 0B	Remote Start Car†	No	No	
BCM	04 0E	Car Alarm Honk†	No	No	
Radio	83 32 ... 00 00	Ticking Sound	No	Yes	
ECM	AE 0E ... 00 7E	Kill Engine	No	Yes	

Table V. Other Example Packets. This table shows packets, their recipients, and their effects that we discovered via observation and reverse-engineering. In contrast to the DeviceControl packets in Tables II, V-A and IV, these packets may be sent during normal operation of the car; we simply exploited the broadcast nature of the CAN bus to send them from CARSHARK instead of their normal sources. For this reason, we did not test most of them at the runway, since they are naturally "tested" during normal operation.
†As ordinarily done by the key fob.

all cylinders simultaneously (even with the car's wheels spinning at 40 MPH when on jack stands), and disable the engine such that it knocks excessively when restarted, or cannot be restarted at all. Additionally, we can forge a packet with the "airbag deployed" bit set to disable the engine. Finally, we also discovered a packet that will adjust the engine's idle RPM.

Brakes. Our fuzzing of the Electronic Brake Control Module (see Table IV) allowed us to discover how to lock individual brakes and sets of brakes, notably without needing to unlock the EBCM with its DeviceControl key. In one case, we sent a random packet which not only engaged the front left brake, but locked it resistant to manual override even through a power cycle and battery removal. To remedy this, we had to resort to continued fuzzing to find a packet that would reverse this effect. Surprisingly, also without needing to unlock the EBCM, we were also able to release the brakes and prevent them from being enabled, even with car's wheels spinning at 40 MPH while on jack stands.

HVAC. We were able to control the cabin environment via the HVAC system: we discovered packets to turn on and off the fans, the A/C, and the heat, in some cases with no manual override possible.

Generic Denial of Service. In another set of experiments, we disabled the communication of individual components on the CAN bus. This was possible at arbitrary times, even with the car's wheels spinning at speeds of 40 MPH when up on jack stands. Disabling communication to/from the ECM when the wheels are spinning at 40 MPH reduces the car's reported speed immediately to 0 MPH. Disabling communication to/from the BCM freezes the instrument panel cluster in its current state (e.g., if communication is disabled when the car is going 40 MPH, the speedometer will continue to report 40 MPH). The car can be turned off in this state, but without re-enabling communication to/from the BCM, the engine cannot be turned on again.

Thus, we were able to easily prevent a car from turning on. We were also able to prevent the car from being turned off: while the car was on, we caused the BCM to activate its ignition output. This output is connected in a wired-OR configuration with the ignition switch, so even if the switch is turned to off and the key removed, the car will still run. We can override the key lock solenoid, allowing the key to be removed while the car is in drive, or preventing the key from being removed at all.

C. Road Testing

Comprehensive and safe testing of these and other attacks requires an open area where individuals and property are at minimal risk. Fortunately, we were able to obtain access to the runway of a de-commissioned airport to re-evaluate many of the attacks we had identified with the car up on jack stands. To maximize safety, we used a second, chase car in addition to the experimental vehicle; see Figure 7. This allowed us to have all but one person outside of the experimented-on car. The experimented-on car was controlled via a laptop running CARSHARK and connected to the CAN bus via the OBD-II port. We in turn controlled this laptop remotely via a wireless link to another laptop in the chase car. To maintain the wireless connection between the laptops, we drove the chase car parallel to the experimented-on car, which also allowed us to capture these experiments on video.

Figure 7. Road testing on a closed course (a de-commissioned airport runway). The experimented-on car, with our driver wearing a helmet, is in the background; the chase car is in the foreground. Photo courtesy of Mike Haslip.

Our experimental protocol was as follows: we started the cars down the runway at the same time, transmitted one or more packets on the experimented-on car's CAN network (indirectly through a command sent from the laptop in the chase car), waited for our driver's verbal confirmation/description (using walkie-talkies to communicate between the cars), and then sent one or more cancellation packets. Had something gone wrong, our driver would have yanked on a cord attached to the CAN cable and pulled the laptop out of the OBD-II. As we verified in preparatory safety tests, this disconnect would have caused the car to revert back to normal within a few seconds; fortunately, our driver never needed to make use of this precaution.

Our allotted time at the airport prevented us from re-verifying all of our attacks while driving, and hence we experimentally re-tested a selected subset of those attacks; the final column of Tables II, V-A, IV, and V contain a check mark for the experiments that we re-evaluated while driving. Most our results while driving were identical to our results on jack stands, except that the EBCM needed to be unlocked to issue DeviceControl packets when the car was traveling over 5 MPH. This a minor caveat from an actual attack perspective; as noted earlier, attack hardware attached to the car's CAN bus can recover the credentials necessary to unlock the EBCM.

Even at speeds of up to 40 MPH on the runway, the attack packets had their intended effect, whether it was honking the horn, killing the engine, preventing the car from restarting, or blasting the heat. Most dramatic were the effects of DeviceControl packets to the Electronic Brake Control Module (EBCM) — the full effect of which we had previously not been able to observe. In particular, we were able to release the brakes and actually *prevent* our driver from braking; no amount of pressure on the brake pedal was able to activate the brakes. Even though we expected this effect, reversed it quickly, and had a safety mechanism in place, it was still a frightening experience for our driver. With another packet, we were able to instantaneously lock the brakes unevenly; this could have been dangerous at higher speeds. We sent the same packet when the car was stationary (but still on the closed road course), which prevented us from moving it at all even by flooring the accelerator while in first gear.

These live road tests are effectively the "gold standard" for our attacks as they represent realistic conditions (unlike our controlled stationary environment). For example, we were never able to completely characterize the brake behavior until the car was on the road; the fact that the back wheels were stationary when the car was on jack stands provided additional input to the EBCM which resulted in illogical behavior. The fact that many of these safety-critical attacks are still effective in the road setting suggests that few DeviceControl functions are actually disabled when the car is at speed while driving, despite the clear capability and intention in the standard to do so.

VI. Multi-Component Interactions

The previous section focused on assessing what an attacker might be able to do by controlling individual devices. We now take a step back to discuss possible scenarios in which multiple components are exploited in a composite attack. The results in this section emphasize that the issue of vehicle security is not simply a matter of securing individual components; the car's network is a heterogeneous environment of interacting components, and must be viewed and secured as such.

A. Composite Attacks

Numerous composite attacks exist. Below we describe a few that we implemented and experimentally verified.

Speedometer. In one attack, we manipulate the speedometer to display an arbitrary speed or an arbitrary offset of the current speed — such as 10 MPH less than the actual speed (halving the displayed speed up to a real speed of 20 MPH in order to minimize obvious anomalies to the driver). This is a composite attack because it requires both intercepting actual speed update packets on the low speed CAN bus (sent by the BCM) and transmitting maliciously-crafted speed update packets with the falsified speed. Such an attack could, for example, trick a driver into driving too fast. We implemented this attack both as a CarShark module and as custom firmware for the AVR-CAN board. The custom firmware consisted of 105 lines of C code. We tested this attack by comparing the displayed speed of one of our cars with the car's actual speed while driving on a closed course and measuring the speed with a radar gun.

Lights Out. Our analysis in Section V uncovered packets that can disable certain interior and exterior lights on the car. We combined these packets to disable *all* of the car's lights when the car is traveling at speeds of 40 MPH or more, which is particularly dangerous when driving in the dark. This includes the headlights, the brake lights, the auxiliary lights, the interior dome light, and the illumination of the instrument panel cluster and other display lights inside the car. This attack requires the lighting control system to be in the "automatic" setting, which is the default setting for most drivers. One can imagine this attack to be extremely dangerous in a situation where a victim is driving at high speeds at night in a dark environment; the driver would not be able to see the the road ahead, nor the speedometer, and people in other cars would not be able to see the victim car's brake lights. We conducted this experiment on both cars while they were on jack stands and while driving on a closed course.

Self-Destruct. Combining our control over various BCM components, we created a "Self-Destruct" demo in which a 60-second count-down is displayed on the Driver Information Center (the dash), accompanied by clicks at an increasing rate and horn honks in the last few seconds. In our demo, this sequence culminated with killing the engine and activating the door lock relay (preventing the occupant from using the electronic door unlock button). This demo, which we tested on both cars, required fewer than 200 lines of code added to CarShark, most of them for timing the clicking and the count-down. One could also extend this sequence to include any of the other actions we learned how to control: releasing or slamming the brakes, extinguishing the lights, locking the doors, and so on.

B. Bridging Internal CAN Networks

Multiple components — including a wealth of aftermarket devices like radios — are attached to or could be attached to a car's low-speed CAN bus. Critical components, like the EBCM brake controller, are connected to the separate high-speed bus, with the Body Control Module (BCM) regulating access between the two buses. One might therefore assume that the devices attached to the low-speed bus, including aftermarket devices, will not be able to adversely impact critical components on the high-speed bus.

Our experiments and analyses found this assumption to be false. Our car's telematics unit is also connected to both buses. We were able to successfully reprogram

our car's telematics unit from a device connected to the car's low-speed bus (in our experiments, a laptop running CARSHARK). Once reprogrammed, our telematics unit acts as a bridge, relaying packets from the low-speed bus onto the high-speed bus. This demonstrates that any device attached to the low-speed bus can bypass the BCM gateway and influence the operation of the safety-critical components. Such a situation is particularly concerning given the abundance of potential aftermarket add-ons available for the low-speed bus. Our complete attack consisted of only the following two steps: initiate a re-programming request to the telematics unit via the low-speed bus; and then upload 1184 bytes of binary code (291 instructions) to the telematics unit's RAM via the low-speed bus.

C. Hosting Code; Wiping Code

This method for injecting code into our car's telematics unit, while sufficient for demonstrating that a low-speed CAN device could compromise a high-speed CAN device via the telematics unit, is also limiting. Specifically, while that attack code is running, the telematics service is not. A more sophisticated attack could implant malicious code *within* the telematics environment itself (either in RAM or by re-flashing the unit). Doing so would allow the malicious code to co-exist with the existing telematics software (we have built such code in the lab). The result provides the attack software with a rich Unix-like environment (our car's telematics unit uses the QNX Neutrino Real-Time Operating System) and provides standard interfaces to additional hardware capabilities (e.g., GPS, audio capture, cellular link) and software libraries (e.g., OpenSSL).

Hosting our own code within a car's ECU enables yet another extension to our attacks: complicating detection and forensic evaluations following any malicious action. For example, the attack code on the telematics unit could perform some action (such as locking the brakes after detecting a speed of over 80 MPH). The attack code could then erase any evidence of its existence on the device. If the attack code was installed per the method described in Section VI-B, then it would be sufficient to simply reboot the telematics unit, with the only evidence of something potentially amiss being the lack of telematics records during the time of the attack. If the attack code was implanted within the telematics environment itself, then more sophisticated techniques may be necessary to erase evidence of the attack code's existence. In either case, such an attack could complicate (or even prevent) a forensic investigation of a crash scene. We have experimentally verified the efficacy of a safe version of this attack while driving on a runway: after the car reaches 20 MPH, the attack code on the telematics unit forces the car's windshield fluid pump and wipers on. After the car stops, the attack code forces the telematics unit to reboot, erasing any evidence of its existence.

VII. DISCUSSION AND CONCLUSIONS

Although we are not the first to observe that computerized automotive systems may present new risks, our empirical approach has given us a unique perspective to reflect on the actual vulnerabilities of modern cars as they are built and deployed today. We summarize these findings here and then discuss the complex challenges in addressing them within the existing automotive ecosystem.

- *Extent of Damage.* Past work, e.g., [19], [24], [26], [27], [28], discuss potential risks to cyber-physical vehicles and thus we knew that adversaries *might* be able to do damage by attacking the components within cars. We did not, however, anticipate that we would be able to directly manipulate safety critical ECUs (indeed, *all* ECUs that we tested) or that we would be allowed to create unsafe conditions of such magnitude.
- *Ease of Attack.* In starting this project we expected to spend significant effort reverse-engineering, with non-trivial effort to identify and exploit each subtle vulnerability. However, we found existing automotive systems — at least those we tested — to be tremendously fragile. Indeed, our simple fuzzing infrastructure was very effective and to our surprise, a large fraction of the random packets we sent resulted in changes to the state of our car. Based on this experience, we believe that a fuzzer itself is likely be a universal attack for disrupting arbitrary automobiles (similar to how the "crashme" program that fuzzed system calls was effective in crashing operating systems before the syscall interface was hardened).
- *Unenforced Access Controls.* While we believe that standard access controls are weak, we were surprised at the extent to which the controls that *did* exist were frequently unused. For example, the firmware on an ECU controls all of its critical functionality and thus the standard for our car's CAN protocol variant describes methods for ECUs to protect against unauthorized firmware updates. We were therefore surprised that we could load firmware onto some key ECUs, like our telematics unit (a critical ECU) and our Remote Control Door Lock Receiver (RCDLR), without any such authentication. Similarly, the protocol standard also makes an earnest attempt to restrict access to DeviceControl diagnostic capabilities. We were therefore also surprised to find that critical ECUs in our car would respond to DeviceControl packets without authentication first.
- *Attack Amplification.* We found multiple opportunities for attackers to amplify their capabilities — either in reach or in stealth. For example, while the designated

gateway node between the car's low-speed and high-speed networks (the BCM) should not expose any interface that would let a low-speed node compromise the high-speed network, we found that we could maliciously bridge these networks through a compromised telematics unit. Thus, the compromise of *any* ECU becomes sufficient to manipulate safety-critical components such as the EBCM. As more and more components integrate into vehicles, it may become increasingly difficult to properly secure all bridging points.

Finally, we also found that, in addition to being able to load custom code onto an ECU via the CAN network, it is straightforward to design this code to completely erase any evidence of itself after executing its attack. Thus, absent any such forensic trail, it may be infeasible to determine if a particular crash is caused by an attack or not. While a seemingly minor point, we believe that this is in fact a very dangerous capability as it minimizes the possibility of any law enforcement action that might deter individuals from using such attacks.[2]

In reflecting on our overall experiences, we observe that while automotive components are clearly and explicitly designed to safely tolerate *failures* — responding appropriately when components are prevented from communicating — it seems clear that tolerating *attacks* has not been part of the same design criteria. Given our results and the observations thus far, we consider below several potential defensive directions and the tensions inherent in them.

To frame the following discussion, we once again stress that the focus of this paper has been on analyzing the security implications *if* an attacker is able to maliciously compromise a car's internal communication's network, not on *how* an attacker might be able to do so. While we can demonstrably access our car's internal networks via several means (e.g., via devices physically attached to the car's internal network, such as a tiny "attack iPod" that we implemented, or via a remote wireless vulnerability that we uncovered), we defer a complete consideration of entry points to future work. Although we consider some specific entry points below (such as malicious aftermarket components), our discussion below is framed broadly and seeks to be as agnostic as possible to the potential entry vector.

Diagnostic and Reflashing Services. Many of the vulnerabilities we discovered were made possible by weak or unenforced protections of the diagnostic and reflashing services. Because these services are never intended for use during normal operation of the vehicle, it is tempting to address these issues by completely locking down such capabilities after the car leaves manufacturing. While it is clearly unsafe for arbitrary ECUs to issue diagnostic and reflashing commands, locking down these capabilities ignores the needs of various stakeholders.

For instance, individuals desire and should be able to do certain things to tune their own car (but not others). Similarly, how could mechanics service and replace components in a "locked-down" automotive environment? Would they receive special capabilities? If so, which mechanics and why should they be trusted? Consider the recently proposed "Motor Vehicle Owners' Right to Repair Act" (H.R. 2057), which would require manufacturers to provide diagnostic information and tools to vehicle owners and service providers, and to provide information to aftermarket tool vendors that enables them to make functionally-equivalent tools. The motivation for this legislation is clear: encouraging healthy competition within the broader automotive industry. Even simple security mechanisms (including some we support, such as signed firmware updates) can be at odds with the vision of the proposed legislation. Indeed, providing smaller and independent auto shops with the ability to service and diagnose vehicles without letting adversaries co-opt those same abilities appears to be a fundamental challenge.

The core problem is lack of access control for the use of these services. Thus, we see desirable properties of a solution to be threefold: arbitrary ECUs should not be able to issue diagnostic and reflashing commands, such commands can only be issued with some validation, and physical access to the car should be required before issuing dangerous commands.

Aftermarket Components. Even with diagnostic and reflashing services secured, packets that appear on the vehicle bus during normal operation can still be spoofed by third-party ECUs connected to the bus. Today a modern automobile leaves the factory containing multiple third-party ECUs, and owners often add aftermarket components (like radios or alarms) to their car's buses. This creates a tension that, in the extreme, manifests itself as the need to either trust all third-party components, or to lock down a car's network so that no third-party components — whether adversarial or benign — can influence the state of the car.

One potential intermediate (and backwards compatible) solution we envision is to allow owners to connect an external filtering device between an untrusted component (such as a radio) and the vehicle bus to function as a trusted mediator, ensuring that the component sends and receives only approved packets.

Detection Versus Prevention. More broadly, certain considerations unique to cyber-physical vehicles raise the possibility of security via detection and correction of anomalies, rather than prevention and locking down of capabilities.

For example, the operational and economic realities of automotive design and manufacturing are stringent. Manufacturers must swiftly integrate parts from different suppliers

[2] As an aside, the lack of a strong forensic trail also creates the possibility for a driver to, after an accident, blame the car's computers for driver error.

(changing as needed to second and third source suppliers) in order to quickly reach market and at low cost. Competitive pressures drive vendors to reuse designs and thus engenders significant heterogeneity. It is common that each ECU may use a different processor and/or software architecture and some cars may even use different communications architectures — one grafted onto the other to integrate a vendor assembly and bring the car to market in time. Today the challenges of integration have become enormous and manufacturers seek to reduce these overheads at all costs — a natural obstacle for instituting strict security policies.

In addition, many of an automobile's functions are safety critical, and introducing additional delay into the processing of, say, brake commands, may be unsafe.

These considerations raise the possibility of exploring the tradeoff between preventing and correcting malicious actions: if rigorous prevention is too expensive, perhaps a quick reversal is sufficient for certain classes of vulnerabilities. Several questions come with this approach: Can anomalous behavior be detected early enough, before any dangerous packets are sent? Can a fail-safe mode or last safe state be identified and safely reverted to? It is also unclear what constitutes abnormal behavior on the bus in the first place, as attacks can be staged entirely with packets that also appear during normal vehicle operation.

Toward Security. These are just a few of many potential defensive directions and associated tensions. There are deep-rooted tussles surrounding the security of cyber-physical vehicles, and it is not yet clear what the "right" solution for security is or even if a single "right" solution exists. More likely, there is a spectrum of solutions that each trade off critical values (like security vs. support for independent auto shops). Thus, we argue that the future research agenda for securing cyber-physical vehicles is not *merely* to consider the necessary technical mechanisms, but to also inform these designs by what is feasible practically and compatible with the interests of a broader set of stakeholders. This work serves as a critical piece in the puzzle, providing the first experimentally guided study into the real security risks with a modern automobile.

Acknowledgments

We thank Mike Haslip, Gary Tomsic, and the City of Blaine, Washington, for their support and for providing access to the Blaine decommissioned airport runway and Mike Haslip specifically for providing Figure 7. We thank Ingolf Krueger for his guidance on understanding automotive architectures, Cheryl Hile and Melody Kadenko for their support on all aspects of the project, and Iva Dermendjieva, Dan Halperin, Geoff Voelker and the anonymous reviewers for comments on earlier versions of this paper. Portions of this work was supported by NSF grants CNS-0722000, CNS-0831532, CNS-0846065, CNS-0905384, CNS-0963695, and CNS-0963702, by a MURI grant administered by the Air Force Office of Scientific Research, by a CCC-CRA-NSF Computing Innovation Fellowship, by a Marilyn Fries Endowed Regental Fellowship, and by an Alfred P. Sloan Research Fellowship.

References

[1] Autosar: Automotive open system architecture. http://www.autosar.org/.

[2] A. Bellissimo, J. Burgess, and K. Fu. Secure software updates: Disappointments and new challenges. In *Proceedings of HotSec 2006*, pages 37–43. USENIX, July 2006.

[3] S. Bono, M. Green, A. Stubblefield, A. Juels, A. D. Rubin, and M. Szydlo. Security analysis of a cryptographically-enabled RFID device. In P. McDaniel, editor, *Proceedings of USENIX Security 2005*. USENIX, Aug. 2005.

[4] Bureau of Transportation Statistics. National Transportation Statistics (Table 1-11: Number of U.S. Aircraft, Vehicles, Vessels, and Other Conveyances), 2008. http://www.bts.gov/publications/national_transportation_statistics/html/table_01_11.html.

[5] CAMP Vehicle Safety Communications 2 Consortium. Cooperative intersection collision avoidance system limited to stop sign and traffic signal violations midterm phase i report, Oct. 2008. Online: http://www.nhtsa.dot.gov/staticfiles/DOT/NHTSA/NRD/Multimedia/PDFs/Crash%20Avoidance/2008/811048.pdf.

[6] CAMP Vehicle Safety Communications 2 Consortium. Vehicle safety communications — applications first annual report, Sept. 2008. Online: http://www.intellidriveusa.org/documents/09042008-vsc-a-report.pdf.

[7] CAMP Vehicle Safety Communications Consortium. Vehicle safety communications project task 3 final report, Mar. 2005. Online: http://www.intellidriveusa.org/documents/vehicle-safety.pdf.

[8] R. Charette. This car runs on code. Online: http://www.spectrum.ieee.org/feb09/7649, Feb. 2009.

[9] DARPA. Grand challenge. http://www.darpa.mil/grandchallenge/index.asp.

[10] A. Edwards. Exclusive: Twitter integration coming to OnStar. Online: http://www.gearlive.com/news/article/q109-exclusive-twitter-integration-coming-to-onstar/, Mar. 2009.

[11] T. Eisenbarth, T. Kasper, A. Moradi, C. Paar, M. Salmasizadeh, and M. Manzuri Shalmani. On the power of power analysis in the real world: A complete break of the KeeLoq code hopping scheme. In D. Wagner, editor, *Proceedings of Crypto 2008*, volume 5157 of *LNCS*, pages 203–20. Springer-Verlag, Aug. 2008.

[12] P. Eisenstein. GM Hy-Wire drive-by-wire hybrid fuel cell vehicle. Online: http://www.popularmechanics.com/automotive/new_cars/1266806.html, Aug. 2002.

[13] B. Emaus. Hitchhiker's Guide to the Automotive Embedded Software Universe, 2005. Keynote Presentation at SEAS'05 Workshop, available at: http://www.inf.ethz.ch/personal/pretscha/events/seas05/bruce_emaus_keynote_050521.pdf.

[14] A. Goodwin. Ford Unveils Open-Source Sync Developer Platform. Online: http://reviews.cnet.com/8301-13746_7-10385619-48.html, Oct. 2009.

[15] T. Hoppe, S. Kiltz, and J. Dittmann. Security threats to automotive CAN networks – practical examples and selected short-term countermeasures. In *SAFECOMP*, 2008.

[16] S. Indesteege, N. Keller, O. Dunkelman, E. Biham, and B. Preneel. A practical attack on KeeLoq. In N. Smart, editor, *Proceedings of Eurocrypt 2008*, volume 4965 of *LNCS*, pages 1–18. Springer-Verlag, Apr. 2008.

[17] ISO. *ISO 11898-1:2003 - Road vehicles – Controller area network*. International Organization for Standardization, Geneva, Switzerland, 2003.

[18] F. Kargl, P. Papadimitratos, L. Buttyan, M. Müter, E. Schoch, B. Wiedersheim, T.-V. Thong, G. Calandriello, A. Held, A. Kung, and J.-P. Hubaux. Secure vehicular communication systems: implementation, performance, and research challenges. *IEEE Communications Magazine*, 46(11):110–118, 2008.

[19] U. E. Larson and D. K. Nilsson. Securing vehicles against cyber attacks. In *CSIIRW '08: Proceedings of the 4th annual workshop on Cyber security and information intelligence research*, pages 1–3, New York, NY, USA, 2008. ACM.

[20] M. Mansur. TunerPro - Professional Automobile Tuning Software. http://www.tunerpro.net/.

[21] M. Melosi. The automobile and the environment in American history. Online: http://www.autolife.umd.umich.edu/Environment/E_Overview/E_Overview.htm, 2004.

[22] S. Mollman. From cars to TVs, apps are spreading to the real world. Online: http://www.cnn.com/2009/TECH/10/08/apps.realworld/, Oct. 2009.

[23] L. E. M. Systems. PCLink - Link ECU Tuning Software. http://www.linkecu.com/pclink/PCLink.

[24] P. R. Thorn and C. A. MacCarley. A spy under the hood: Controlling risk and automotive EDR. *Risk Management*, February 2008.

[25] Virginia Tech Transportation Institute. Intersection collision avoidance — violation task 5 final report, Apr. 2007. Online: http://www.intellidriveusa.org/documents/final-report-04-2007.pdf.

[26] M. Wolf, A. Weimerskirch, and C. Paar. Security in automotive bus systems. In *Proceedings of the Workshop on Embedded Security in Cars 2004*, 2004.

[27] M. Wolf, A. Weimerskirch, and T. Wollinger. State of the art: Embedding security in vehicles. *EURASIP Journal on Embedded Systems*, 2007.

[28] Y. Zhao. Telematics: safe and fun driving. *Intelligent Systems, IEEE*, 17(1):10–14, Jan/Feb 2002.