Securing Embedded Systems: Analyses of Modern Automotive Systems and Enabling Near-Real Time Dynamic Analysis

Karl Koscher

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2014

Reading Committee:

Tadayoshi Kohno, Chair

Gaetano Borriello

Shwetak Patel

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Securing Embedded Systems:
From Analyses of Modern Automotive Systems to Enabling Dynamic Analysis

Karl Koscher

Chair of the Supervisory Committee:
Associate Professor Tadayoshi Kohno
Department of Computer Science and Engineering

Today, our life is pervaded by computer systems embedded inside everyday products. These *embedded systems* are found in everything from cars to microwave ovens. These systems are becoming increasingly sophisticated and interconnected, both to each other and to the Internet. Unfortunately, it appears that the security implications of this complexity and connectivity have mostly been overlooked, even though ignoring security could have disastrous consequences; since embedded systems control much of our environment, compromised systems could be used to inflict physical harm.

This work presents an analysis of security issues in embedded systems, including a comprehensive security analysis of modern automotive systems. We hypothesize that dynamic analysis tools would quickly discover many of the vulnerabilities we found. However, as we will discuss, there are several challenges in applying traditional dynamic analysis tools to embedded systems. We propose and evaluate new tools to overcome these challenges.

Table of Contents

Acknowledgements

This dissertation would not be possible without the innumerable contributions, mentorship, and support of others. While it is impossible to thank everyone who contributed to this work or my personal development, there are several people I would like to explicitly acknowledge.

First, I'd like to thank my advisor, Tadayoshi Kohno, for his invaluable mentorship, advice, and support throughout my career as a PhD student. Yoshi has been a veritable fountain of cool and interesting ideas and provided the support to see those crazy ideas through to completion.

Second, I'd like to thank my collaborators and co-authors of the automotive security analysis projects in Chapters 2 and 3. This was a huge undertaking that required a large team of awesome collaborators. At the University of Washington, this includes Alexei Czeskis, Franzi Roesner, Shwetak Patel, Tadayoshi Kohno, and Conrad Meyer. At the University of California San Diego, this includes Stephen Checkoway, Damon McCoy, Danny Anderson, Hovav Shacham, and Stefan Savage.

Several others also helped make the automotive research successful. I'd like to thank Mike Haslip, Gary Tomsic, and the City of Blaine, Washington, for their support and for providing access to the Blaine decommissioned airport runway. I'd like to explicitly thank Mike Haslip, Chief of the Blaine Police Department, for documenting our work through photos and providing Figure 4. I'd like to thank Ingolf Krueger for his guidance on understanding automotive architectures. I'd like to thank Cheryl Hile and Melody Kadenko for their support on all aspects of the automotive research projects, including procuring the vehicles, equipment, and test courses, as well as helping us deal with university bureaucracies. I'd like to thank Iva Dermendjieva, Dan Halperin, Geoff Voelker for

their feedback on Chapter 2, and Dan Wallach for his feedback on Chapter 3 as well as shepherding the corresponding paper for USENIX Security.

I'd like to thank David Molnar for his ideas and feedback on the automatic security analyses portion of my thesis. David is a leading expert in automated white-box fuzz testing, having built not one but two white-box fuzz testing systems (SAGE and SmartFuzz). His ideas and feedback were extremely influential in the design of the automated analysis system introduced in Chapter 4.

Over my tenure as a PhD student, I've collaborated with a ton of awesome people on a variety of projects. While these projects are not part of my dissertation, they nevertheless have helped make me the computer scientist I am today. In particular, I'd like to thank Adrienne Andrew, Max Andrews, Yaw Anokwa, Jacob Appelbaum, Magdalena Balazinska, Gaetano Borriello,Vjeko Brajkovic, Tanzeem Choudhury, Sunny Consolvo, Tamara Denning, Ian Finder, Dieter Fox, Batya Friedman, Nell Carden Grey, Steven D. Gribble, Dirk Haehnel, Beverly Harrison, Carl Hartung, Bruce Hemmingway, Jeffrey Hightower, Ari Juels, Predrag "Pedja" Klasnja, Anthony LaMarca, James A. Landay, Louis LeGrand, Jonathan Lester, Cynthia Matuszek, Brian Mayton, Ali Rahimi, Marsh Ray, Adam Rea, Bruce Schneier, Joshua R. Smith, Emad Soroush, David J. St. Hilaire, Evan Welbourne, and Danny Wyatt. My sincerest apologies to anyone I might have missed.

I'd like to thank the other members of the UW CSE Security and Privacy Research lab, including lab Iva Dermendjieva, Miro Enev, Adam Lerner, Amit Levy, Peter Ney, Temitope Oluwafemi, Alisha Saxena, Ian Smith, Alex Takakuwa, and Paul Vines, for their friendship, support, and ideas.

There are a few other people I'd especially like to thank. First, I'd like to thank Gaetano Borriello for his excellent classes which introduced me to hardware and embedded systems, for getting me started in research as his Research Engineer, for championing my decision to go to grad school, and for his invaluable career advice. Thank you for everything! I'd like to thank Waylon Brunette for being Gaetano's awesome TA and getting me up to speed on various hardware projects. I'd also like to thank Lindsay Michimoto for her support and help in all sorts of administrative and personal matters, and especially for her assistance getting me in to the UW CSE PhD program.

Finally, I'd like to thank the awesome group of friends I've acquired throughout my time in the program, including Alexei Czeskis, Tamara Denning, Alice Neels, Kathleen Tuite, and especially Cynthia Matuszek (all from the incoming class of 2007, a.k.a. "The StalinGrads"); your friendship and support have kept me sane during my time as a PhD student!

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 DARPA grant FA8750-12-2-0107, by the SHARPS ARRA grant from the U.S. Department of Health and Human Services, by a CCC-CRA-NSF Computing Innovation Fellowship, by a Marilyn Fries Endowed Regental Fellowship, by an Alfred P. Sloan Research Fellowship, and by a Ford Motor Company Fellowship.

Dedication

To my parents, Annetta and Walter Koscher, whose love and support have enabled me to fulfill my dreams.

Chapter 1: Introduction and Background

1.1. Introduction

Today, our life is pervaded by computer systems *embedded* inside everyday products. These *embedded systems* are found in everything from cars to microwave ovens. Historically, these systems have had dedicated functionality, tightly coupled with the products they are embedded in. For example, an embedded system inside a microwave oven may be as simple as a program that turns on the magnetron (RF source) for a set amount of time and monitors the door for safety.

However, these systems are becoming increasingly sophisticated and interconnected, both to each other and to public networks like the Internet. Unfortunately, it appears that the security implications of this complexity and connectivity have mostly been overlooked, even though ignoring security could have disastrous consequences. Since embedded systems control much of our environment, compromised systems could be used to inflict physical harm. We are already seeing state-sponsored malware developed to do so, such as with Stuxnet, which reprogrammed the firmware of Iran's uranium enrichment centrifuge controllers, causing the centrifuges to self-destruct [**1**]. Even without physical harm, massively-exploited devices can irreparably damage a manufacturer's reputation, as was almost the case just before Microsoft announced their Trustworthy Computing initiative [**2**] [**3**].

Therefore, in my research, I aim to understand 1) what kinds of vulnerabilities exist in modern embedded systems, 2) why such vulnerabilities exist, 3) what tools may be appropriate to discover and correct these vulnerabilities, and 4) the effectiveness of these tools.

To understand the nature of embedded systems vulnerabilities, my colleagues and I perform a thorough security analysis of a modern automobile and report our findings in Chapters 2 and 3. Aside from being interesting cyber-physical systems, modern automobiles have dozens of embedded computers, ranging in complexity from simple 8-bit microcontrollers with a few kilobytes of RAM to fully-featured, multicore, 32-bit systems, running POSIX-compliant real-time operating systems. A thorough security analysis of automobiles exposes us to a wide range of embedded systems, allowing us to identify common vulnerability themes.

Based on our findings in Chapters 2 and 3, we believe that many vulnerabilities in modern embedded systems could be discovered and eliminated using dynamic analysis techniques *if* they could be used against these systems. However, we find that there are significant challenges in applying traditional dynamic analysis tools to embedded systems. Therefore, in Chapter 4 we build and evaluate a new hardware/software tool to overcome some of these challenges and enable near-real-time dynamic analysis of embedded systems.

1.2. Background

Many of the contributions made in this dissertation rely on the low-level details of embedded systems. However, today most computer scientists do not have to concern themselves with the specifics of hardware implementation. Therefore, we provide a brief overview of some important concepts in embedded systems in Section 1.2.1.

There are over 250 million registered passenger automobiles in the United States [**4**], and the vast majority of these are computer controlled to a significant degree. 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 science research community. In Section 1.2.2, we provide basic background context concerning automotive embedded systems architecture in general.

1.2.1. Embedded Systems

Embedded systems are now the *dominant* form of the computing, vastly outnumbering "traditional" computers such as PCs. However, there is a huge amount of diversity among embedded systems. Some use simple 8-bit microcontrollers with less than 1 kB of memory. Some use 64-bit multicore processors. However, there are some salient features that embedded systems share. First, they are generally special-purpose devices, as opposed to general-purpose PCs[1]. Second, they are tightly coupled with their environment—taking input from a variety of sensors, and affecting the environment through its outputs.

Many embedded systems are implemented with *microcontrollers*, as opposed to microprocessors. Whereas microprocessors need external components to operate (such as RAM and I/O devices), microcontrollers integrate a CPU with these components onto a single chip. Pressure to make devices smaller and more capable has led to many microcontrollers implementing a huge variety of peripherals, such as audio, video, and USB interfaces. A single highly-capable microcontroller is sufficient to build an entire system; for this reason, such microcontrollers are often called *Systems on Chip* (or *SoCs*). Most SoCs today use some type of ARM processor. However, the peripheral interfaces can vary widely.

[1] This distinction is blurring, though. For example, mobile phones used to be dedicated systems that only supported phone calls. Today, the phone functionality is only one small aspect of a complex system with installable apps and Internet access.

Peripherals are controlled by the firmware through *Memory Mapped I/O* (*MMIO*). With MMIO, peripheral registers are assigned memory addresses and are accessed as if they were memory. While MMIO can be technically sufficient to fully control a peripheral, usually some optimizations are made. For example, if a program is waiting for input from a serial port, it can repeatedly *poll* the serial controller's status register until input is received. Polling has many downsides, but in the context of embedded systems, perhaps the biggest problem with polling is that it wastes energy by keeping the CPU running in a tight loop.[2] As an alternative, peripherals can raise *interrupts*. As the name implies, this interrupts the normal path of execution and transfer control to an interrupt handler. The interrupt handler can then respond to the peripheral and transfer control back to the originally-executing code.

Another optimization is called *Direct Memory Access* (DMA).With DMA, large chunks of memory can be transferred between peripherals and RAM using dedicated logic. This frees the processor to perform other tasks. In some cases, DMA is handled by a DMA controller, which can copy memory between arbitrary addresses (such as RAM and a peripheral's MMIO registers). In other cases, the peripheral itself can initiate transactions on the system bus; this is called *bus mastering*.

1.2.2. Automobiles

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, gear-shift, 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

[2] Transistor switching is the main power draw in modern CMOS ICs. Temporarily halting CPUs while waiting for an event can dramatically cut the amount of power consumed.

driver, and the passengers. Indeed, one 2009 estimate suggests that the typical luxury sedan now contains over 100MB 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* [**5**] [**6**].

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. Originally intended to verify emissions-related systems, the OBD-II port is now used by a myriad of aftermarket products. Many of these products, such as Automatic [**7**] and usage-based insurance dongles [**8**], act as a wireless gateway into cars' networks. User-upgradable subsystems such as audio players are also 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 [**9**] while Ford announced that it will open its Sync telematics system as a platform for third-party applications [**10**]. Finally, proposed future vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2X) communications systems [**11**] [**12**] [**13**] [**14**] will only broaden the attack surface further.

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.

(1) 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 [**15**]. 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 [**5**].

(2) 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 newer vehicles 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 announced that GM's OnStar will offer integration with Twitter [**16**]). Even full "steer-by-wire" functionality has been seen in a range of concept cars including GM's widely publicized Hy-wire fuel cell vehicle [**17**].

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*) [**18**] and FlexRay [**19**], and software technology platforms (cf. Autosar [**20**]) 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 power-train 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.

(3) 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 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.

1.3. Related work

There are four principal areas of related work: security analyses of automotive systems, "classic" (non-automated) security analyses of other embedded systems, dynamic analyses of programs for security, and attempts to automatically analyze embedded systems.

1.3.1. Automotive Systems

We are not the first to observe the potential fragility of the automotive environment. In the academic context and prior to our work, several groups described potential vulnerabilities in

automotive systems, e.g., [**21**] [**22**] [**23**] [**24**] [**25**]. 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., [**26**], most of these efforts considered threats abstractly; considering "what-if" questions about a hypothetical attacker. Part of our 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.

Furthermore, prior to our work there was very little understanding of the *external* attack surface of modern vehicles. Among the exceptions is Rouf et al.'s analysis of the wireless Tire Pressure Monitoring System (TPMS) in a modern vehicle [**27**]. While their work was primarily focused on the privacy implications of TPMS broadcasts, they also described methods for manipulating drivers by spoofing erroneous tire pressure readings and, most relevant to our work, an experience in which they accidentally caused the ECU managing TPMS data to stop functioning through wireless signals alone. Still others have focused on the computer security issues around car theft, including Francillon et al.'s recent demonstration of relay attacks against keyless entry systems [**28**], and the many attacks on the RFID-based protocols used by engine immobilizers to identify the presence of a valid ignition key, e.g., [**29**], [**30**], [**31**]. Orthogonally, there has been work that considers the *future* security issues (and expanded attack surface) associated with proposed vehicle-to-vehicle (V2V) systems (sometimes also called vehicular ad-hoc networks, or VANETs) [**13**] [**32**] [**33**]. To the best of our knowledge, however, we are the first to consider the full external attack surface of the contemporary automobile, characterize the threat models under which this surface is exposed, and experimentally demonstrate the practicality of remote threats, remote control, and remote data exfiltration. Our experience further gives us the vantage point to reflect on some of the ecosystem

challenges that give rise to these problems and point the way forward to better secure the automotive platform in the future.

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 [**34**], [**35**]. These groups are not malicious; 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.

Following initial publication of our results in 2010 [**36**] and 2011 [**37**], there has been a growing interest in automotive systems in the broader security and "maker" communities. In 2013, Miller and Valasek obtained results similar to those in Chapter 2 on different vehicles [**38**] and released the tools they used for doing so. In 2014, Miller and Valasek analyzed the potential remote attack surfaces of twenty different vehicles [**39**]. Several web sites dedicated to vehicle network reverse engineering have emerged, such as CanBusHack [**40**] and the Vehicle Reverse Engineering Wiki [**41**]. The *Hack a Day* web site has a good four-part introduction on CAN bus hacking [**42**]. A number of hardware projects have been started as well, including the OpenXC platform [**43**] (sponsored by Ford), the GoodThopter [**44**], and the CANBus Triple [**45**].

Finally, we point out that while there is an emerging effort focused on designing fully autonomous vehicles (e.g., DARPA Grand Challenge [**46**] and Google's self-driving car [**47**]), these are specifically designed to be robotically controlled. While such vehicles would undoubtedly introduce yet new

security concerns, in our work we concern ourselves solely with the vulnerabilities in today's commercially-available automobiles.

1.3.2. Security Analyses of Embedded Systems

As embedded systems have become more complex and connected, the security issues facing them have evolved. First, we will discuss some of the classic embedded systems issues, mostly relating to keeping cryptographic secrets safe from physical attackers. Then, we will discuss some of the threats emerging as embedded systems become more capable. A discussion of automated analyses of embedded systems appears in Section 1.3.4

(1) Classic Embedded Systems Security

Early work focused on threats where the attacker was physically present, as this was the only way to interact with many embedded systems. Around this time, smart cards were becoming adopted for a wide range of applications, including GSM SIM cards, prepaid phone cards, and pay TV. In [**48**], Anderson and Kuhn describe some early attacks against smart cards, including non-invasive fault injection attacks and physical probing attacks. Later, in 2004, [**49**] provided an overview of the challenges in designing secure embedded systems. It includes a long discussion of earlier work on fault injection, timing side-channels, and power side-channels.

(a) Fault Injection

Fault injection attacks involve intentionally inducing errors into computations. Typically this is done by glitching the power supply or clock lines. This temporarily brings some components (such as CPU registers) outside their rated operating ranges (such as voltage or timing constraints), which can induce faults. This is normally done for one of two reasons. First, faults can cause security logic to

fail, allowing an attacker to trick a CPU into going down a code path that is not normally allowed. For example, if a smart card is verifying a PIN, an attacker could induce a fault to make the PIN check succeed regardless of whether the PIN is correct. Second, faults can produce mathematical errors that may be exploitable. For example, inducing faults in certain implementations of the RSA signing algorithm can produce erroneous signatures that can be used to trivially derive the private key [**50**].

(b) Side-Channel Attacks

Side-channel attacks involve recovering information about private data used in a computation leaked through unintended side effects. For example, a side channel attack against an encryption system may leak the key, or at least information to narrow the set of possible keys. Side channels can leak information in many forms: acoustic [**51**], optical [**52**], electromagnetic, etc. Regardless of form, side channels of *computations* (as opposed to physical processes, like key presses and printing) reveal information about either timing or power. Both can be leveraged for practical attacks.

Timing analysis relies on the fact that naïve cryptographic algorithms take a varying amount of time depending on their inputs. In [**53**], Kocher introduced the first theoretical timing attack on cryptographic algorithms. His attack focuses on algorithms using square-and-multiply modular exponentiation, including implementations of RSA, Diffe-Hellman, and DSS. The attack works by observing the execution time of a number of modular exponentiations with known bases and modulus. If an attacker can model the execution time taken by the first n iterations of each modular exponentiation, then she can decrease the variance of the execution times collected by subtracting the timing model of the first n iterations from them. A correct timing model depends on knowing the secret exponent, but if the attacker is only modeling the first n iterations, then she only needs to

know the least-significant *n* bits. By guessing the next bit and checking to see if the variance decreases, an attacker can successively determine each bit of the exponent. Kocher's work was subsequently improved several times, including by Brumley and Boneh, who in [**54**] demonstrated that timing attacks work remotely and against other RSA optimizations used in the popular OpenSSL [**55**] library.

Power analysis relies on the fact that in modern CMOS ICs, the amount of power consumed during a clock cycle is in large part determined by the number of transistors changing state. Therefore, the amount of power consumed depends on the computations performed. In [**56**], Kocher describes two ways this can be exploited. The first, called Simple Power Analysis (SPA), relies on certain computations having distinct power signatures. For example, public-key algorithms that rely on square-and-multiply modular exponentiation may leak their private key if the squaring and multiplying operations can be differentiated. The second, called Differential Power Analysis (DPA), uses simple statistical techniques to obtain information leaked through noisy power traces. In DPA, the attacker collects power traces over many cryptographic operations. The attacker makes a hypothesis about a sub-computation (such as the value of one of the key bytes being fed into an S-Box) and classifies the traces based on a measurable result of that sub-computation (such as the LSB of the S-Box's output). If the hypothesis is incorrect, the average trace of each class will be indistinguishable from each other. If the hypothesis is correct, the average trace of each class will differ significantly at points where power consumption is highly correlated with the hypothesis. [**57**] provides a thorough and accessible introduction to both SPA and DPA, and provides an example of breaking AES with DPA as well as breaking RSA with SPA.

(2) Emerging threats

While these threats remain important for certain applications (such as smart cards and set-top boxes), new threats have emerged as the progression of Moore's law has enabled embedded devices to become more complex. As Kocher points out [**56**] (and we find in our own work), vulnerabilities often appear at boundaries of abstraction. For example, cryptographers often assume error- and side-channel-free computation, but in reality, their algorithms run on imperfect, physical hardware. As another example, C programmers oblivious to how the stack works can end up writing trivial buffer overflow vulnerabilities. As embedded systems become more complex, it becomes necessary to create additional abstractions to manage this complexity, which creates more opportunities for vulnerabilities. To investigate what kinds of vulnerabilities might emerge from this added complexity, we and other researchers have performed security analyses on several new types of complex embedded devices.

(a) Voting Machines

An early but influential example of this is [**58**], which audited the (accidentally leaked) source code to the Diebold AccuVote-TS direct recording electronic (DRE) voting system. The authors found almost no regard for security in its design. While the system used smartcards to authorize ballot casting and administration, the protocol did not incorporate any cryptography, allowing an attacker to forge smartcards to cast additional ballots and perform administrative functions. Configuration data is not authenticated, allowing an attacker to change audit counters and ballot definitions. Votes are stored in-order and encrypted poorly, with a fixed key and IV, potentially allowing an attacker to modify votes and link voters with their votes. Results are transmitted back to the central tabulator with no encryption or authentication.

While the AccuVote-TS is a complex system, new techniques can uncover vulnerabilities in older, simpler systems as well. In [**59**], Checkoway et al. demonstrated that return-oriented programming could be used to exploit a Sequoia AVC Advantage voting machine, which was designed in the early 1980s and was purposefully designed to execute code only in ROM. This work demonstrates one of the challenges of securing embedded systems: they must be able to withstand attacks for their entire service lifetime, which may be as long as several decades.

(b) Consumer Devices

In [**60**], Saponas et al. looked at three new consumer ubiquitous computing products (the Slingbox Pro, Nike+iPod Sport Kit, and Microsoft Zune) to gauge the state of privacy and security practices built into these types of products. All three products had privacy issues that could have been addressed with more careful development.

The Slingbox Pro is a device that lets users view television remotely over the Internet. Content is encrypted, possibly to prevent eavesdroppers from learning anything about what is being streamed (although legal concerns are a more likely reason). However, by using a variable bit-rate compression codec, the Slingbox Pro leaks identifying information about the content being streamed. After 10 minutes of streaming, the authors were able to identify which movie (of 26) was being viewed with 62% accuracy.

The Nike+iPod Sport Kit allows users to track data about their walks and runs, such as duration, step count, and intensity. It works by embedding sensors into specially-designed shoes, which broadcast sensor readings to compatible iPod receivers. The sensors broadcast unique IDs, which allows the receivers to filter out other users' sensors. However, these IDs are not protected in any

way, and can be eavesdropped from up to 20 meters away. The authors developed a proof-of-concept surveillance device which could be deployed in several locations to track Nike+iPod users throughout the deployment.

The Microsoft Zune media player had an interesting feature: it would let you share content with anyone within WiFi range. To mitigate abuse, the Zune allows users to block senders. However, the authors discovered this was trivial to bypass.

(c) Medical Devices

In [**61**], Halperin et al. analyzed the wireless protocol used by a popular implantable cardioverter defibrillator (ICD). Several privacy and integrity issues were discovered. Private patient data, including name, therapy, and sometimes even their social security number, can be exfiltrated or intercepted from an ICD. Additionally, the authors were able to reverse-engineer enough of the protocol to change certain settings, including the clock and therapies. Perhaps most concerning was their ability to deliver command shocks that would induce fibrillation, even with therapies disabled. Noting that ICDs have extremely limited power budgets (ICDs have non-rechargeable batteries that can only be surgically replaced), the authors presented three zero-power defenses built on top of the WISP platform [**62**]. The first uses RF power from the reader to alert the patient that their ICD is being interrogated. The second offloads authentication to the WISP, which is able to perform a cryptographic handshake with the reader using only the reader's RF power. The third communicates an ephemeral key through acoustic waves transmitted through the body by a reader. For each of these defenses, the authors experimentally verified that sufficient power can be harvested from the reader by testing them through a bag of bacon and ground beef, which serves as a good human analogue.

(d) Cross-Channel Scripting Attacks

In [**63**], Bojinov, Bursztein, and Boneh demonstrate a new type of scripting attack against networked devices such as network-attached storage devices, lights-out management devices, digital picture frames, and cell phones. Whereas traditional cross-site scripting attacks are caused by a web application accepting JavaScript code and reflecting it back to the victim, cross-channel scripting attacks accept JavaScript code from non-web sources, such as CIFS file shares, and include this code in their web UI (e.g., a view of the file system). While our work focuses on lower-level attacks, using a taint-tracking dynamic analysis tool in conjunction with the system we introduce in Chapter 4 may be able to detect these vulnerabilities.

(e) Large-Scale Analysis of Embedded Firmwares

In [**64**], Costin, Zaddach, Francillon, and Balzarotti crawl the web and collect almost 760,000 firmware updates for a variety of embedded systems. Approximately 32,000 of these were automatically unpacked and analyzed for easy-to-identify vulnerabilities, such as weak default passwords, SSH and other backdoors, fixed private RSA keys, vulnerable HTTP server configurations, and known vulnerable software signatures. Thirty-eight previously-unknown vulnerabilities were discovered affecting over 123 different products. One of the primary limitations of their system is its inability to analyze the code itself, either statically or dynamically. Thus, only simple and easy-to-identify vulnerabilities (such as hardcoded passwords) are identified.

(f) Non-Academic Results

Outside of academia, embedded systems security is receiving increasing amounts of attention as well. A few of the more notable examples are discussed in this section.

In 2005, Lynn presented a talk at Black Hat USA on developing Cisco IOS shellcode [**65**], which he claimed could severely cripple the Internet. He did so even after Cisco pressured his employer and Black Hat to forbid him from speaking about the vulnerabilities. Cisco subsequently filed suits against Lynn and Black Hat [**66**], but reached an agreement where all materials related to the talk were turned over to Cisco [**67**].

At Black Hat USA 2010, Jack demonstrated vulnerabilities in two different ATMs which "jackpotted" (dumped all of the money inside) those ATMs [**68**]. One ATM provided an USB port, accessible with a key easily purchased online, that allowed unauthenticated firmware updates to be installed. The other ATM had vulnerabilities in its remote update mechanism.

Video game consoles are attractive targets for hackers. These consoles are typically locked down so that only licensed and authentic games can be played. However, there are groups of enthusiasts who like to develop and run "homebrew" software—unlicensed games or other software that are developed by hobbyists. Doing so requires bypassing security mechanisms built in to these consoles. There are countless exploits for most systems, from simple buffer overflows to signature comparison exploits [**69**] and even cryptographic breaks resulting from insufficient randomness [**70**].

Finally, it is nearly impossible to avoid discussions of "cyber terrorism" attacks on critical infrastructure. While there is a lot of speculation about the potential threats, solid facts are hard to come by. At Blackhat Federal 2006, two security consultants presented several real-world vulnerabilities they've encountered during their pen-testing services [**71**], including several control networks connected to the Internet and embedded control (SCADA) systems full of vulnerabilities.

(3) Reflection

With the increasing complexity and connectedness of embedded systems, we see attacks moving up the stack. For example, the firmware in early smart cards was simple enough that one could have a reasonable level of assurance that there were no bugs in its logic, and thus attackers focused on lower-level (e.g., physical and side-channel) exploits. With increased complexity, it is more difficult to retain that level of assurance, and indeed there is a lot of prior work which have demonstrated exploits of firmware-level bugs. However, due to the heterogeneity of embedded devices, this work has largely been domain-specific, focusing on individual types of devices. While parts of this dissertation focuses on another domain—automobiles—we do so in part to gain insight into issues affecting a large variety of embedded systems, as automobiles incorporate a large number of diverse embedded systems. We use this insight to guide our development of a generic tool, introduced in Chapter 4, which enables dynamic analysis across many different types of embedded systems.

1.3.3. Dynamic Analysis

There has been a great deal of prior work proposing various dynamic analyses techniques to discover and fix security vulnerabilities. Dynamic analysis is useful in cases where the system is too complex to model or analyze statically, which is rapidly becoming the case with most embedded systems. Additionally, since dynamic analyses work against real systems (rather than models), additional unexpected behavior may be uncovered. Dynamic analysis methods also have the nice property of minimizing false positives. Whereas static analysis can tell you that a problem *may* exist, dynamic analyses will point to concrete problems.

(1) Fuzz Testing

An early example of dynamic analysis is *fuzz testing*, invented after Miller noticed that line noise on his dial-up connection would sometimes cause remote programs to crash [**72**]. Miller et al. automated generation of "line noise" and applied it to several UNIX utilities, finding that they were able to crash between 25% and 33% of standard utilities included in various UNIX OSes. From these experiments, they were able to find common errors and classified each crash according to its cause.

While seemingly powerful, naïve fuzz testing is unable to catch many errors. The approach in [**72**] simply generated random characters. If we want to fuzz file parsers, this approach is unlikely to work since there will usually be at least some sanity checks on the file format. Fortunately, there have been several advances in fuzzing techniques, which can be broadly classified as either *black-box* or *white-box*.

(a) Black-Box Fuzz Testing

Black-box fuzz testing involves intelligently crafting inputs without knowledge of the target program's internals. This is usually done with one of two approaches: *mutational* and *generational*. Mutational fuzzing takes a valid input and randomly perturbs it. While this approach is able to find vulnerabilities that completely random inputs would not trigger [**73**], it is unlikely to find vulnerabilities which depend on a deep knowledge of the input format. Generational fuzzers, such as SPIKE [**74**] and the Peach Fuzzing Platform [**75**], use input grammars and constraints specified by users to intelligently create inputs. Since these inputs are mostly valid, they will often pass most sanity checks, allowing them to discover subtle flaws.

(b) White-Box Fuzz Testing

White-box fuzz testing relies on observing program execution to make intelligent choices about what to fuzz. As a simplified example, consider an *if* statement that checks whether *input[0] == 42*. Traditional black-box fuzzers do not have the ability to observe this behavior, so they are unlikely to generate inputs that exercise both code paths from that branch unless explicitly encouraged to via a user-specified grammar (such as defining *input[0]* to be 42 or not). There are a few closely related papers in this area, all using symbolic execution in one form or another.

EXE [**76**] treats all user input initially as unconstrained symbolic variables. Instructions that only use concrete operands are executed normally. However, for instructions that use one or more symbolic operands, EXE produces a symbolic, constrained result. For example, if x is an unconstrained input, and the statement $y := x + 10$ is executed, then y becomes symbolic and constrained to be $x + 10$. When a branch based on a symbolic value is executed (such as checking whether $y == 42$), the execution is forked and constraints for each side of the branch are added (e.g., $y == 42$ and $y != 42$). If an execution crashes, the constraints are solved to generate the corresponding input.

DART [**77**], which was independently developed, uses a similar approach. However, the target program is run both concretely and symbolically. Instead of forking execution on symbolic branching conditions, DART simply constrains the value so that symbolic execution follows the concrete execution. The set of constraints built up over an execution is called the *path constraint*. By negating a branching constraint in the path constraint (and dropping subsequent constraints), DART solves for new inputs to explore additional code paths.

SAGE [**78**] builds on the work of DART to apply this technique to large programs, such as media parsers, compression codecs, and even parts of Microsoft Office 2007. The primary difference between DART and SAGE is the search algorithm—DART uses a depth-first approach, while SAGE uses a generational approach. Child executions are ranked by block coverage heuristics, and the highest-ranked execution becomes the parent for the next generation.

(2) Other types of dynamic analysis

Of course, fuzz testing is not the only type of dynamic analysis, and we believe the work presented in Chapter 4 applies to much more than fuzz testing. For example, taint tracking (an overview of which is provided by [**79**]) is useful for checking for unintentional information flow (such as from untrusted user input to sensitive state). Run-time memory verification systems like Valgrind's Memcheck [**80**] are powerful tools that can detect memory leaks, access to unallocated or freed memory, or buffer overflows. They do so by tracking the state of each memory chunk. In particular, Memcheck works by tracking whether a memory chunk has been allocated and whether it contains valid data.

1.3.4. Automated Analyses of Embedded Systems

The poor state of embedded security and the seriousness of its consequences have led researchers to propose new ways to *automatically* analyze embedded systems, building on the success of traditional dynamic analysis tools. However, there are a number of challenges in applying traditional dynamic analysis tools to embedded systems.

Whereas traditional software is written against OS-provided APIs, the "API" that firmware is written against is usually a hardware specification. Peripherals typically expose their behavior

through several *memory-mapped* registers. These registers appear as normal memory, but reads and writes to these addresses directly control the hardware. With the large heterogeneity of embedded devices, faithfully reproducing hardware behavior to dynamic analysis tools is a time-consuming and error-prone proposition.

FIE [**81**] symbolically executes the firmware of small, MSP430-based embedded devices. FIE overcomes the challenges in the diversity of devices and the need to understand peripheral semantics by treating *all* peripheral I/O as an unconstrained symbolic input. Unfortunately, this can easily lead to a state space explosion, making this technique impractical for all but the smallest embedded systems.

Avatar [**82**] attempts to constrain the number of states explored by using the *actual hardware* as a guide for peripheral semantics. It does so by redirecting peripheral I/O to the real device, either by using a JTAG debugger or through serial communication with an in-memory stub loaded onto the target. Unfortunately, with the ability to do only about five memory operations per second, redirecting all I/O is prohibitively slow. Avatar overcomes this limitation by migrating executing code between the emulator and the device, and emulating only small portions of interest of the firmware. However, this optimization is unsuitable for timing-sensitive systems, such as medical devices with watchdogs in coprocessors. In our work, which we started independently and in parallel with Avatar, we seek to overcome this limitation by enabling *real-time* peripheral interaction.

Chapter 2: Security Analysis of Internal Automotive Interfaces[3]

2.1. Introduction and Threat Model

In this chapter we focus on what an attacker could do to a car *if* she was able to maliciously communicate on the car's internal network. We intentionally and explicitly skirt the question of a "threat model". That said, this does raise 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 Chapter 3, 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 Section 2.5.3, 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. We wish to be clear that vulnerabilities in such services are not purely theoretical. In Chapter 3, we demonstrate the ability to

[3] This chapter is based on "Experimental Security Analysis of a Modern Automobile," published and presented at the IEEE Symposium on Security and Privacy (Oakland) 2010. DOI: 10.1109/SP.2010.34

remotely compromise key ECUs in our car via externally-facing vulnerabilities, amplify the impact of these remote compromises using the results in this chapter, and ultimately monitor and control our car remotely over the Internet.

2.2. Experimental Setup

Our experimental analyses focus on two 2009 automobiles of the same make and model.[4] 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 1 lists some of the most important ECUs in our car.

2.2.1. Experimental Environments

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

[4] We believe the risks identified in this chapter arise from the *architecture* of modern automobiles 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.

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.

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 engine using information from sensors to determine the amount of fuel, ignition timing, and other engine parameters.		✓
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 pre-tensioners.	✓	
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 1: Key Electronic Control Units (ECUs) within our cars, their roles, and which CAN buses they are on.

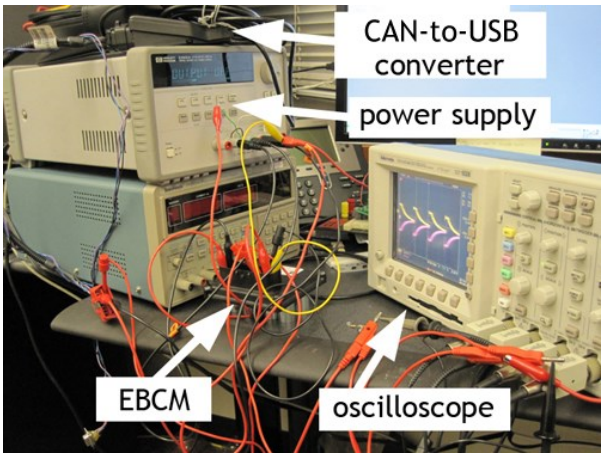

Figure 1: *Example bench setup within our lab.* The Electronic Brake Control Module (EBCM) is hooked up to a power supply, a CAN-to-USB converter, and an oscilloscope.

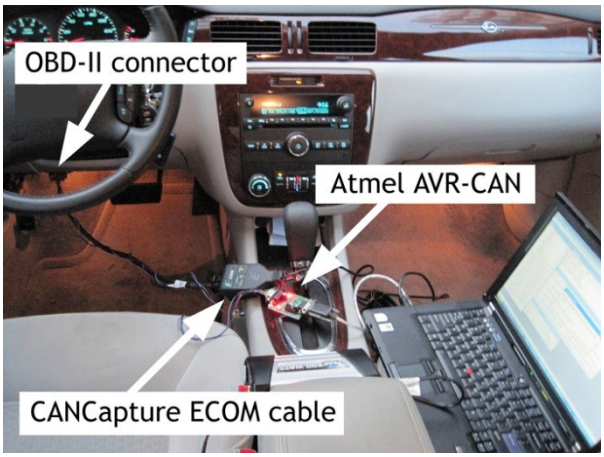

Figure 2: *Example experimental setup.* The laptop is running our custom CARSHARK CAN network analyzer and attack tool. The laptop is connected to the car's OBD-II port.

- **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).

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

Figure 4: *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.

- **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 4).

  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.

2.2.2. The CARSHARK Tool

To facilitate our experimental analysis, we wrote CARSHARK—a custom CAN bus analyzer and packet injection tool (see Figure 5). While commercially available CAN sniffers exist, 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 does not 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.

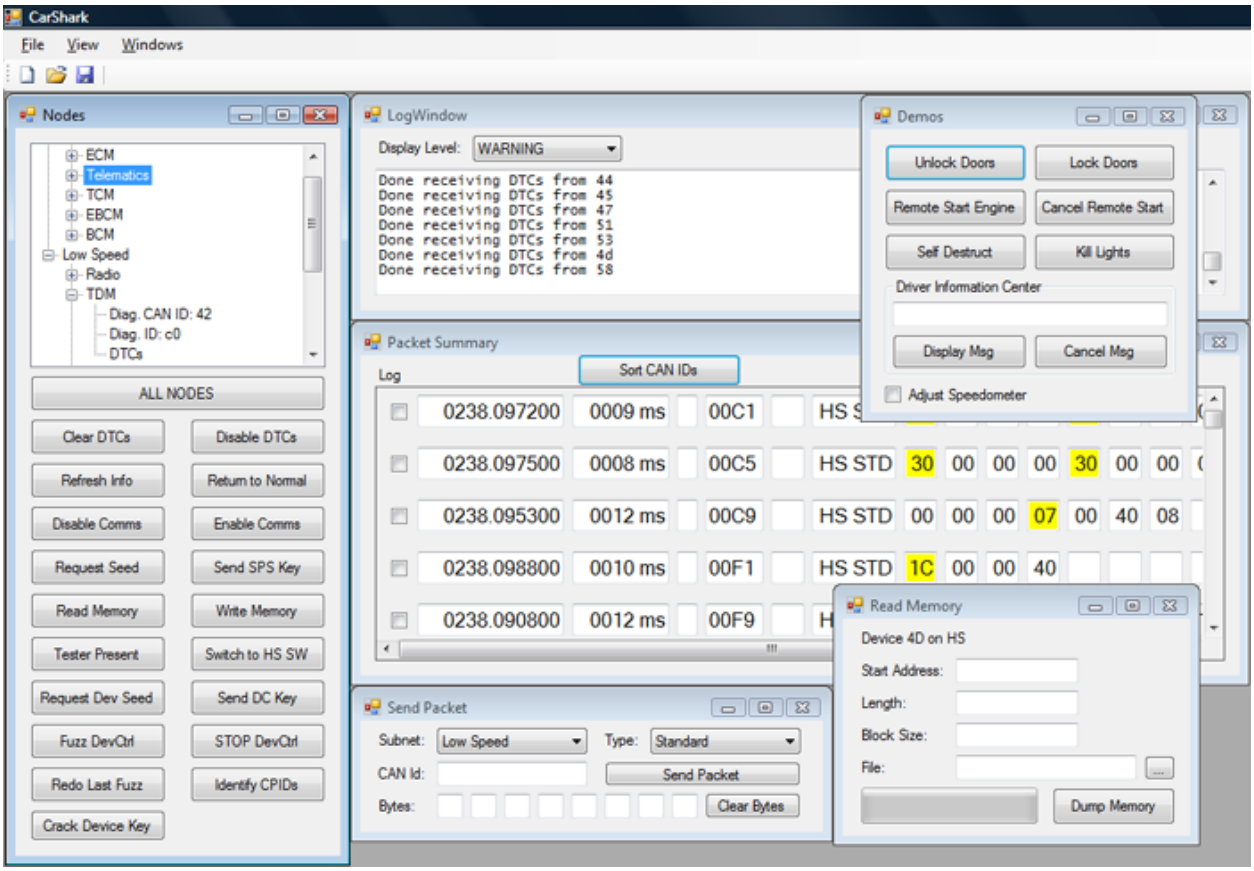

Figure 5: *Screenshot of the CarShark Interface.* CarShark is being used to sniff the CAN bus. Values that have recently changed are highlighted in yellow. The left panel lists all recognized nodes on the high and low speed subnets of the CAN bus and has some action buttons. The demo panel on the right provides some proof-of-concept demos.

2.3. 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 manufacturer's standards.

2.3.1. 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 [**18**]) 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, Volkswagen, Toyota, and others).

A CAN packet (shown in Figure 6) 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.

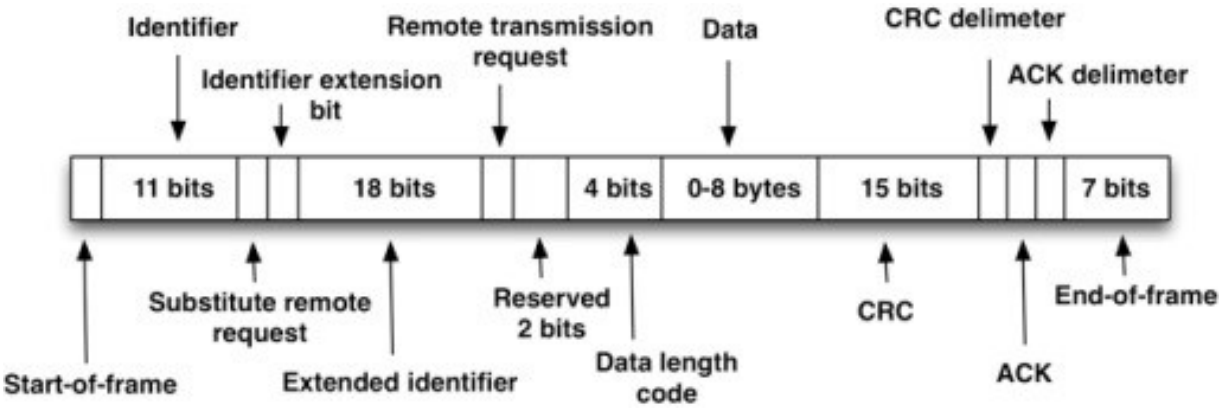

Figure 6: *CAN packet structure*. Extended frame format is shown; base frame format is similar.

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 h*igh-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 the two buses. Finally, the manufacturer's protocol standards define a range of services to be implemented by ECUs.

2.3.2. CAN Security Challenges

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

(1) 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.

(2) Fragility to Denial of Service

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.

(3) 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 2.4.

(4) 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 manufacturer standard states "under no circumstances shall the encryption algorithm ever reside in the node." (The tester, however, does have the ability to compute the key, either through knowledge of the algorithm or a connection to an Internet service.) 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 manufacturer's 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 power-cycling 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.

(5) 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 member of our research team 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 [**83**]. 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 2.3.3 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.

2.3.3. 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.

(1) 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 1 even with the car's wheels moving at speed on jack stands and while driving on the closed road course.

(2) 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.

(3) 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 2.3.2(4)). 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 noncompliance, 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.

(4) 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 2, 3, and 4.

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 2: Body Control Module (BCM) DeviceControl Packet Analysis. This table shows BCM DeviceControl packets and their effects that we discoveredduring fuzz testing with one of our cars on jack stands. A X 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 3: Engine Control Module (ECM) DeviceControl Packet Analysis. This table is similar to Table 2.

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 4: Electronic Brake Control Module (EBCM) DeviceControl Packet Analysis. This table is similar to Table 2.
†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.

Dest. 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 5: 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 2, 3 and 4, 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.
† As ordinarily done by the key fob.

(5) 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.

2.4. 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 2.5. We omit certain details (such as complete packet payloads) to prevent would-be attackers from using our results directly.

2.4.1. Attack Methodology

Recall that Table 1 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.

(1) 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.

(2) 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, brake lights, 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.

(3) 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.

2.4.2. 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, and in many cases immobilized on jack stands

for safety, as shown in Figure 3. Some of our results are summarized in Tables 2, 3, and 4 for fuzzing, and in Table 5 for other results. Tables 2, 3, and 4 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 2.4.3. Table 5 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 (although road tests were only performed with the first car due to logistical constraints, e.g., finding a suitable closed course).

(1) 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 5 presents some of these results.

(2) 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 5). For example, Figure 7 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 2.5.1(1).

Figure 7: Displaying an arbitrary message and a false speedometer reading on the Driver Information Center. Note that the car is in Park.

(3) 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 5) and by fuzzing packets on the high-speed

bus (as summarized in Table 2), 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.

(4) Engine

Most of the attacks against the engine were found by fuzzing DeviceControl requests to the ECM. These findings are summarized in Table 3. We were able to boost the engine RPM temporarily, disturb engine timing by resetting the learned crankshaft angle sensor error, disable 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.

(5) Brakes

Our fuzzing of the Electronic Brake Control Module (see Table 4) 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.

(6) 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.

(7) 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.

2.4.3. 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 4. 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.

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 disconnected the laptop from the OBD-II port. 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 2, 3, 4, and 5 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 standards to do so.

2.5. 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.

2.5.1. Composite Attacks

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

(1) 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.

(2) Lights Out

Our analysis in Section 2.4 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 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.

(3) 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.

2.5.2. 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 reprogramming 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.

2.5.3. Hosting and 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 2.5.2, 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.

2.6. 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., [**21**], [**22**], [**23**], [**24**], [**25**], 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.[5]

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 chapter has been on analyzing the security implications *if* an attacker is able to maliciously compromise a car's internal communications 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 remote wireless vulnerabilities that we uncovered), we defer a complete consideration of entry points to Chapter 3. Although we consider some specific entry points below (such as malicious

[5] 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.

aftermarket components), our discussion below is framed broadly and seeks to be as agnostic as possible to the potential entry vector.

2.6.1. 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 proposed "Motor Vehicle Owners' Right to Repair Act" (H.R. 2057, 2009), 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.

2.6.2. 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.

2.6.3. 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 (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.

2.6.4. 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.

Chapter 3: Security Analysis of External Automotive Interfaces[6]

3.1. Introduction

The threat model underlying Chapter 2 was met with significant, and justifiable, criticism (e.g., [**84**], [**85**], [**86**]). In particular, it was widely felt that presupposing an attacker's ability to *physically* connect to a car's internal computer network may be unrealistic. Moreover, it is often pointed out that attackers with physical access can easily mount non-computerized attacks as well (e.g., cutting the brake lines).[7]

This situation suggests a significant gap in knowledge, and one with considerable practical import. To what extent are external attacks possible, to what extent are they practical, and what vectors represent the greatest risks? Is the etiology of such vulnerabilities the same as for desktop software and can we think of defense in the same manner? This chapter seeks to fill this knowledge gap through a systematic and empirical analysis of the remote attack surface of late model mass-production sedan.

In this chapter, we make four principal contributions:

1. **Threat model characterization.** We systematically synthesize a set of *possible* external attack vectors as a function of the attacker's ability to deliver malicious input via particular modalities: indirect physical access, short-range wireless access, and long-range wireless

[6] This chapter is based on "Comprehensive Security Analysis of Automotive Attack Surfaces," published and presented at the USENIX Security Symposium 2011.

[7] Perhaps the most pithy version of this critique comes from security expert Ken Tindell's comments to *The Register* [**86**]: *"I was utterly shocked to discover that apparently if you prise open an embedded system, reflash its program code, you can pretty much do anything to the I/O connected to the system," he said. "Well knock me down with a feather." … "The only risk they encountered was a theoretical one (viz. that a telematics system connected to the in-vehicle networking could hack the car). It's highly theoretical because the challenges of hacking a car are vastly more than hacking a banking system."*

access. Within each of these categories, we characterize the attack surface exposed in current automobiles and their surprisingly large set of I/O channels.

2. **Vulnerability analysis.** For each access vector category, we investigate one or more concrete examples in depth and assess the level of actual exposure. In each case we find the existence of *practically exploitable vulnerabilities* that permit arbitrary automotive control *without requiring direct physical access*. Among these, we demonstrate the ability to compromise a car via vulnerable diagnostics equipment widely used by mechanics, through the media player via inadvertent playing of a specially modified song in WMA format, via vulnerabilities in hands-free Bluetooth functionality and, finally, by calling the car's cellular modem and playing a carefully crafted audio signal encoding both an exploit and a bootstrap loader for additional remote-control functionality.
3. **Threat assessment.** From these uncovered vulnerabilities, we consider the question of "utility" to an attacker: what capabilities does the vulnerability enable? Unique to this work, we study how an attacker might leverage a car's external interfaces for post-compromise control. We demonstrate multiple post-compromise control channels (including TPMS wireless signals and FM radio), interactive remote control via the Internet and real-time data exfiltration of position, speed and surreptitious streaming of cabin audio (i.e., anything being said in the vehicle) to an outside recipient. Finally, we also explore potential attack scenarios and gauge whether these threats are purely conceptual or whether there are plausible motives that transform them into actual risks. In particular, we demonstrate complete capabilities for both theft and surveillance.
4. **Synthesis.** On reflection, we noted that the vulnerabilities we uncovered have surprising similarities. We believe that these are not mere coincidences, but that many of these security problems arise, in part, from systemic structural issues in the automotive ecosystem. Given

these lessons, we make a set of concrete, pragmatic recommendations which significantly raise the bar for automotive system security. These recommendations are intended to "bridge the gap" until deeper architectural redesign can be carried out.

3.2. Automotive Threat Models

While past work has illuminated specific classes of threats to automotive systems—such as the technical security properties of their internal networks [**21**] [**22**] [**23**] [**24**] [**25**] [**87**]—we believe that it is critical for future work to place specific threats and defenses in the context of the entire automotive platform. In this section, we aim to bootstrap such a comprehensive treatment by characterizing the threat model for a modern automobile. Though we present it first, our threat model is informed significantly by the experimental investigations we carried out, which are described in subsequent sections.

In defining our threat model, we distinguish between *technical* capabilities and *operational* capabilities.

Technical capabilities describe our assumptions concerning what the adversary knows about its target vehicles as well as her ability to analyze these systems to develop malicious inputs for various I/O channels. For example, we assume that the adversary has access to an instance of the automobile model being targeted and has the technical skill to reverse engineer the appropriate subsystems and protocols (or is able to purchase such information from a third-party). Moreover, we assume she is able to obtain the appropriate hardware or medium to transmit messages whose encoding is appropriate for any given channel.[8] When encountering cryptographic controls, we also

[8] For the concrete vulnerabilities we will explore, the hardware cost for such capabilities is modest, requiring only commodity laptop computers, an audio card, a USB-to-CAN interface, and, in a few instances, an inexpensive, off-the-shelf USRP software radio platform.

assume that the adversary is computationally bounded and cannot efficiently brute force large shared secrets, such as large symmetric encryption keys. In general, we assume that the attacker only has access to information that can be directly gleaned from examining the systems of a vehicle similar to the one being targeted.[9] We believe these assumptions are quite minimal and mimic the access afforded to us when conducting this work.

By contrast, operational capabilities characterize the adversary's requirements in delivering a malicious input to a particular access vector in the field. In considering the full range of I/O capabilities present in a modern vehicle, we identify the qualitative differences in the challenges required to access each channel. These in turn can be roughly classified into three categories: indirect physical access, short-range wireless access, and long-range wireless access. In the remainder of this section we explore the threat model for each of these categories and within each we synthesize the "attack surface" presented by the full range of I/O channels present in today's automobiles. Figure 8 highlights where I/O channels exist on a modern automobile today.

3.2.1. Indirect Physical Access

Modern automobiles provide several physical interfaces that either directly or indirectly access the car's internal networks. We consider the full physical attack surface here, under the constraint that the adversary may not *directly* access these physical interfaces herself but must instead work through some intermediary.

[9] A question which we do not consider in this work is the extent to which the attack surface is "portable" between vehicle models from a given manufacturer. There is significant evidence that some such attacks are portable as manufacturers prefer to build a small number of underlying platforms that are specialized to deliver model-specific features, but we are not in a position to evaluate this question comprehensively.

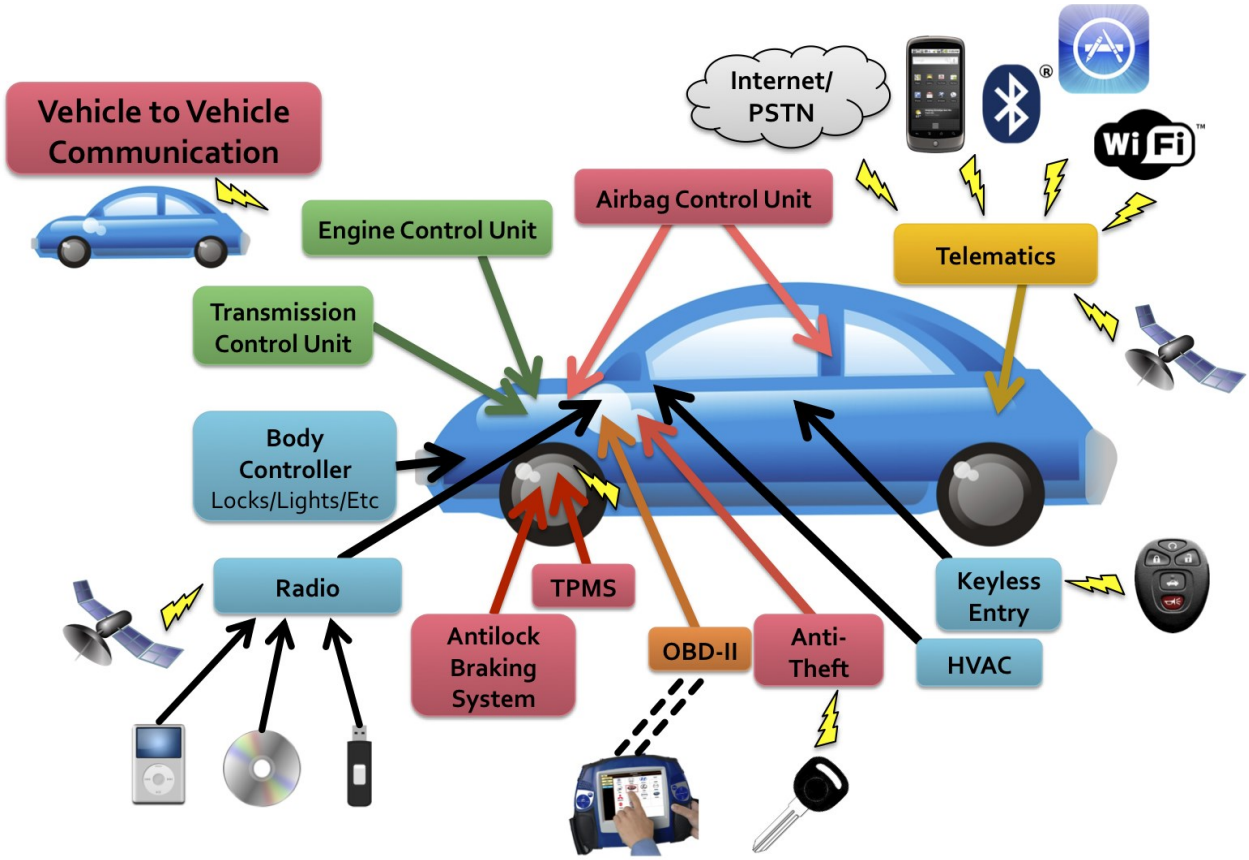

Figure 8: Digital I/O channels appearing on a modern car. Colors indicate rough grouping of ECUs by function.

(1) OBD-II

The most significant automotive interface is the OBD-II port, federally mandated in the U.S., which typically provides direct access to the automobile's key CAN buses and can provide sufficient access to compromise the full range of automotive systems [**87**]. While our threat model forbids the adversary from direct access herself, we note that the OBD-II port is commonly accessed by service personnel during routine maintenance for both diagnostics and ECU programming.

Historically this access is achieved using dedicated handheld "scan" tools such as Ford's NGS, Nissan's Consult II and Toyota's Diagnostic Tester which are themselves programmed via

Windows-based personal computers. For modern vehicles, most manufacturers have adopted an approach that is PC-centric. Under this model, a laptop computer interfaces with a "PassThru" device (typically directly via USB or WiFi) that in turn is plugged into the car's OBD-II port. Software on the laptop computer can then interrogate or program the car's ECUs via this device (typically using the standard SAE J2534 API). Examples of such tools include Toyota's TIS, Ford's VCM, Nissan's Consult 3, and Honda's HDS, among others.

In both situations Windows-based computers directly or indirectly control the data to be sent to the automobile. Thus, if an adversary were able to compromise such systems at the dealership she could amplify this access to attack any cars under service. Such laptop computers are typically Internet-connected (indeed, this is a requirement for some manufacturers' systems), so traditional means of personal computer compromise could be employed.

Further afield, electric vehicles may also communicate with external chargers via the charging cable. An adversary able to compromise the external charging infrastructure may thus be able to leverage that access to subsequently attack any connected automobile.

(2) Entertainment: Disc, USB, and iPod

The other important class of physical interfaces are focused on entertainment systems. Virtually all automobiles shipped today provide a CD player able to interpret a wide variety of audio formats (raw "Red Book" audio, MP3, WMA, and so on). Similarly, vehicle manufacturers also provide some kind of external digital multimedia port (typically either a USB port or an iPod/iPhone docking port) for allowing users to control their car's media system using their personal audio player or phone. Some manufacturers have widened this interface further; BMW and Mini recently announced their

support for "iPod Out," a scheme whereby Apple media devices will be able to control the display on the car's console.

Consequently, an adversary might deliver malicious input by encoding it onto a CD or as a song file and using social engineering to convince the user to play it. Alternatively, she might compromise the user's phone or iPod out of band and install software onto it that attacks the car's media system when connected.

Taking over a CD player alone is a limited threat; but, for a variety of reasons, automotive media systems are not standalone devices. Indeed, many such systems are now CAN bus interconnected, either to directly interface with other automotive systems (e.g., to support chimes, certain hands-free features, or to display messages on the console) or simply to support a common maintenance path for updating all ECU firmware. Thus, counterintuitively, a compromised CD player can offer an effective vector for attacking other automotive components.

3.2.2. Short-Range Wireless Access

Indirect physical access has a range of drawbacks including its operational complexity, challenges in precise targeting, and the inability to control the time of compromise. Here we weaken the operational requirements on the attacker and consider the attack surface for automotive wireless interfaces that operate over short ranges. These include Bluetooth, Remote Keyless Entry, RFIDs, Tire Pressure Monitoring Systems, WiFi, and Dedicated Short-Range Communications. For this portion of the attack surface we assume that the adversary is able to place a wireless transmitter in proximity to the car's receiver (between 5 and 300 meters depending on the channel).

(1) Bluetooth

Bluetooth has become the de facto standard for supporting hands-free calling in automobiles and is standard in mainstream vehicles sold by all major automobile manufacturers. While the lowest level of the Bluetooth protocol is typically implemented in hardware, the management and services component of the Bluetooth stack is often implemented in software. In normal usage, the Class 2 devices used in automotive implementations have a range of 10 meters, but others have demonstrated that this range can be extended through amplifiers and directional antennas [**88**].

(2) Remote Keyless Entry

Today, all but entry-level automobiles shipped in the U.S. use RF-based remote keyless entry (RKE) systems to remotely open doors, activate alarms, flash lights and, in some cases, start the ignition (all typically using digital signals encoded over 315 MHz in the U.S. and 433 MHz in Europe).

(3) Tire Pressure

In the U.S., all 2007 model year and newer cars are required to support a Tire Pressure Monitoring System (TPMS) to alert drivers about under or over inflated tires. The most common form of such systems, so-called "Direct TPMS," uses rotating sensors that transmit digital telemetry (frequently in similar bands as RKEs).

(4) RFID car keys

RFID-based vehicle immobilizers are now nearly ubiquitous in modern automobiles and are mandatory in many countries throughout the world. These systems embed an RFID tag in a key or key fob and a reader in or near the car's steering column. These systems can prevent the car from operating unless the correct key (as verified by the presence of the correct RFID tag) is present.

(5) Emerging Short-Range Channels

A number of manufacturers have started to discuss providing 802.11 WiFi access in their automobiles, typically to provide "hotspot" Internet access via bridging to a cellular 3G/4G data link. In particular, Ford offers this capability in the 2012 Ford Focus. (Several 2011 models also provided WiFi receivers, but we understand they were used primarily for assembly line programming.)

Finally, while not currently deployed, an emerging wireless channel is defined in the Dedicated Short-Range Communications (DSRC) standard, which is being incorporated into proposed standards for Cooperative Collision Warning/Avoidance and Cooperative Cruise Control. Representative programs in the U.S. include the Department of Transportation's Cooperative Intersection Collision Avoidance Systems (CICAS-V) and the Vehicle Safety Communications Consortium's VSC-A project. In such systems, forward vehicles communicate digitally to trailing cars to inform them of sudden changes in acceleration to support improved collision avoidance and harm reduction.

(6) Summary

For all of these channels, if a vulnerability exists in the ECU software responsible for parsing channel messages, then an adversary may compromise the ECU (and by extension the entire vehicle) simply by transmitting a malicious input within the automobile's vicinity.

3.2.3. Long-Range Wireless

Finally, automobiles increasingly include long distance (greater than 1 km) digital access channels as well. These tend to fall into two categories: broadcast channels and addressable channels.

(1) Broadcast Channels

Broadcast channels are channels that are not specifically directed towards a given automobile but can be "tuned into" by receivers on-demand. In addition to being part of the external attack surface, long-range broadcast mediums can be appealing as control channels (i.e., for triggering attacks) because they are difficult to attribute, can command multiple receivers at once, and do not require attackers to obtain precise addressing for their victims.

The modern automobile includes a plethora of broadcast receivers for long-range signals: Global Positioning System (GPS),[10] Satellite Radio (e.g., SiriusXM receivers common to late-model vehicles from Honda/Acura, GM, Toyota, Saab, Ford, Kia, BMW and Audi), Digital Radio (including the U.S. HD Radio system, standard on 2011 Ford and Volvo models, and Europe's DAB offered in Ford, Audi, Mercedes, Volvo and Toyota among others), and the Radio Data System (RDS) and Traffic Message Channel (TMC) signals transmitted as digital subcarriers on existing FM-bands.

The range of such signals depends on transmitter power, modulation, terrain, and interference. As an example, a 5 W RDS transmitter can be expected to deliver its 1.2 kbps signal reliably over distances up to 10 km. In general, these channels are implemented in an automobile's media system (radio, CD player, satellite receiver) which, as mentioned previously, frequently provides access via internal automotive networks to other key automotive ECUs.

[10] We do not currently consider GPS to be a practical access vector for an attacker because in all automotive implementations we are aware of, GPS signals are processed predominantly in custom hardware. By contrast, we have identified significant software-based input processing in other long-range wireless receivers.

(2) Addressable Channels

Perhaps the most important part of the long-range wireless attack surface is that exposed by the remote telematics systems (e.g., Ford's Sync, GM's OnStar, Toyota's SafetyConnect, Lexus' Enform, BMW's BMW Assist, and Mercedes-Benz' mbrace) that provide continuous connectivity via cellular voice and data networks. These systems provide a broad range of features supporting safety (crash reporting), diagnostics (early alert of mechanical issues), anti-theft (remote track and disable), and convenience (hands-free data access such as driving directions or weather).

These cellular channels offer many advantages for attackers. They can be accessed over arbitrary distance (due to the wide coverage of cellular data infrastructure) in a largely anonymous fashion, typically have relatively high bandwidth, are two-way channels (supporting interactive control and data exfiltration), and are individually addressable.

3.2.4. Stepping Back

There is a significant knowledge gap between these possible threats and what is known to date about automotive security. Given this knowledge gap, much of this threat model may seem far-fetched. However, in the next section we find quite the opposite. For each category of access vector we will explore one or two aspects of the attack surface deeply, identify concrete vulnerabilities, and explore and demonstrate practical attacks that are able to completely compromise our target automobile's systems without requiring direct physical access.

3.3. Vulnerability Analysis

We now turn to our experimental exploration of the attack surface. We first describe the automobile and key components under evaluation and provide some context for the tools and methods we

employed. We then explore in-depth examples of vulnerabilities via indirect physical channels (CDs and service visits), short-range wireless channels (Bluetooth), and long-range wireless (cellular). Table 6 summarizes these results as well as our qualitative assessment of the cost (in effort) to discover and exploit these vulnerabilities.

3.3.1. Experimental Context

All of our experimental work focuses on a moderately priced late model sedan with the standard options and components. Between 100,000 and 200,000 of this model were produced in the year of manufacture. The car includes less than 30 ECUs comprising both critical drivetrain components as well as less critical components such as windshield wipers, door locks and entertainment functions. These ECUs are interconnected via multiple CAN buses, bridged where necessary. The car exposes a number of external vectors including the OBD-II port, media player, Bluetooth, wireless TPMS sensors, keyless entry, satellite radio, RDS, and a telematics unit. The last provides voice and data access via cellular networks, connects to all CAN buses, and has access to Bluetooth, GPS and independent hands-free audio functionality (via an embedded microphone in the passenger cabin). We also obtained the manufacturer's standard "PassThru" device used by dealerships and service stations for ECU diagnosis and reprogramming, as well as the associated programming software. For several ECUs, notably the media and telematics units, we purchased a number of identical replacement units via on-line markets to accommodate the inevitable "bricking" caused by imperfect attempts at code injection.

Vulnerability Class	Channel	Implemented Capability	User-Visible	Scale	Full Control	Cost	Section
Direct Physical	OBD-II port	Plug attack hardware directly into car OBD-II port	Yes	Small	Yes	Low	Chapter 2
Indirect physical	CD	CD-based firmware update	Yes	Small	Yes	Medium	3.3.2(1)
	CD	Special song file (WMA)	Yes*	Medium	Yes	Medium-High	3.3.2(1)
	PassThru	WiFi or wired control connection to advertised PassThu devices	No	Small	Yes	Low	3.3.2(2)
	PassThru	WiFi or wired shell injection	No	Viral	Yes	Low	3.3.2(2)
Short-range wireless	Bluetooth	Buffer overflow with paired Android phone or Trojan app	No	Large	Yes	Low-Medium	3.3.3
	Bluetooth	Sniff MAC address, brute force PIN, buffer overflow	No	Small	Yes	Low-Medium	3.3.3
Long-range wireless	Cellular	Call car, authentication exploit, buffer overflow (using laptop)	No	Large	Yes	Medium-High	3.3.4

	Cellular	Call car, authentication exploit, buffer overflow (using iPod with exploit audio file, earphones, and a telephone)	No	Large	Yes	Medium-High	3.3.4

Table 6: *Attack surface capabilities*. The Visible to User column indicates whether the compromise process is visible to the user (the driver or the technician); we discuss social engineering attacks for navigating user detection in the body. For (∗), users will perceive a malfunctioning CD. The Scale column captures the approximate scale of the attack, e.g., the CD firmware update attack is small-scale because it requires distributing a CD to each target car. The Full Control column indicates whether this exploit yields full control over the component's connected CAN bus (and, by transitivity, all the ECUs in the car). Finally, the Cost column captures the approximate effort to develop these attack capabilities.

Building on our previous work, we first established a set of messages and signals that could be sent on our car's CAN bus (via OBD-II) to control key components (e.g., lights, locks, brakes, and engine) as well as injecting code into key ECUs to insert persistent capabilities and to bridge across multiple CAN buses [**87**]. Note, such inter-bus bridging is critical to many of the attacks we explore since it exposes the attack surface of one set of components to components on a separate bus; we explain briefly here. Most vehicles implement multiple buses, each of which host a subset of the ECUs.[11] However, for functionality reasons these buses must be interconnected to support the complex coupling between pairs of ECUs and thus a small number of ECUs are physically connected to multiple buses and act as logical bridges. Consequently, by modifying the "bridge" ECUs (either via a vulnerability or simply by reflashing them over the CAN bus as they are designed to be) an attacker can amplify an attack on one bus to gain access to components on another. Consequently, the result is that compromising any ECU with access to some CAN bus on our vehicle (e.g., the media player) is sufficient to compromise the entire vehicle.

[11] In prior work we hypothesized that CAN buses were purposely separated for security reasons—one for safety-critical components like the radio and engine and the other for less important components such as a radio. Based on discussions with industry experts we have learned that this separation has until now often been driven by bandwidth and integration concerns and not necessarily security.

Combining these ECU control and bridging components, we constructed a general "payload" that we attempted to deliver in our subsequent experiments with the external attack surface.[12] To be clear, **for every vulnerability we demonstrate, we are able to obtain complete control over the vehicle's systems**. We did not explore weaker attacks.

For each ECU we consider, our experimental approach was to extract its firmware and then explicitly reverse engineering its I/O code and data flow using disassembly, interactive logging and debugging tools where appropriate. In most cases, extracting the firmware was possible directly via the CAN bus (this was especially convenient because in most ECUs we encountered, the flash chips are not socketed and while we were able to desolder and read such chips directly, the process was quite painful).

Having the firmware in hand, we performed three basic types of analysis: raw code analysis, in situ observations, and interactive debugging with controlled inputs on the bench. In the first case, we identified the microprocessor (e.g., different components described in this chapter use System on Chip (SoC) variants of the PowerPC, ARM, Super-H and other architectures) and used the industry-standard IDA Pro disassembler to map control flow and identify potential vulnerabilities, as well as debugging and logging options that could be enabled to aid in reverse engineering.[13] In-situ observation with logging enabled allowed us to understand normal operation of the ECU and let us concentrate on potential vulnerabilities near commonly used code paths. Finally, ECUs were removed from the car and placed into a test harness on the bench from which we could carefully

[12] In this work we experimented with two equivalent vehicles to ensure that our results were not tied to artifacts of a particular vehicle instance.
[13] IDA Pro does not support embedded architectures as well as x86 and consequently we needed to modify IDA Pro to correctly parse the full instruction set and object format of the target system. In one particular case (for the TPMS processor) IDA Pro did not provide any native support and we were forced to write a complete architecture module in order to use the tool.

control all inputs and monitor outputs. In this environment, interactive debuggers were used to examine memory and single step through vulnerable code under repeatable conditions. For one such device, the Super-H-based media player, we resorted to writing our own native debugger and exported a control and output interface through an unused serial UART interface we "broke out" off the circuit board.

In general, we made use of any native debugging I/O we could identify. For example, like the media player, the telematics unit exposed an unused UART that we tapped to monitor internal debugging messages as we interactively probed its I/O channels. In other cases, we selectively rewrote ECU memory (via the CAN bus or by exploiting software vulnerabilities) or rewrote portions of the flash chips using the manufacturer-standard ECU programming tools. For the telematics unit, we wrote a new character driver that exported a command shell to its UNIX-like operating system directly over the OBD-II port to enable interactive debugging in a live vehicle. In the end, our experience was that although the ECU environment was somewhat more challenging than that of desktop operating systems, it was surmountable with dedicated effort.

3.3.2. Indirect Physical Channels

We consider two distinct indirect physical vectors in detail: the media player (via the CD player) and service access to the OBD-II port. We describe each in turn along with examples of when an adversary might be able to deliver malicious input.

(1) Media Player

The media player in our car is fairly typical, receiving a variety of wireless broadcast signals, including analog AM and FM as well as digital signals via FM sub-carriers (RDS, called RBDS in the

U.S.) and satellite radio. The media player also accepts standard compact discs (via physical insertion) and decodes audio encoded in a number of formats including raw Red Book audio as well as MP3 and WMA files encoded on an ISO 9660 filesystem.

The media player unit itself is manufactured by a major supplier of entertainment systems, both stock units directly targeted for automobile manufacturers as well as branded systems sold via the aftermarket. Software running on the CPU handles audio parsing and playback requests, UI functions, and directly handles connections to the CAN bus.

We found two vulnerabilities. First, we identified a latent update capability in the media player that will automatically recognize an ISO 9660-formatted CD with a particularly named file, present the user with a cryptic message and, if the user does not press the appropriate button, will then reflash the unit with the data contained therein.[14] Second, knowing that the media player can parse complex files, we examined the firmware for input vulnerabilities that would allow us to construct a file that, if played, gives us the ability to execute arbitrary code.

For the latter, we reverse-engineered large parts of the media player firmware, identifying the file system code as well as the MP3 and WMA parsers. In doing so, we documented that one of the file read functions makes strong assumptions about input length *and* moreover that there is a path through the WMA parser (for handling an undocumented aspect of the file format) that allows arbitrary length reads to be specified; together these allow a buffer overflow.

[14] This is not the standard method that the manufacturer uses to update the media player software and thus we believe this is likely a vestigial capability in the supplier's code base.

This particular vulnerability is not trivial to exploit. The buffer that is overflowed is not on the stack but in a BSS segment, without clear control data variables to hijack. Moreover, immediately after the buffer are several dynamic state variables whose values are continually checked and crash the system when overwritten arbitrarily.

To overcome these and other obstacles, we developed a native in-system debugger that communicates over an unused serial port we identified on the media player. This debugger lets us dump and alter memory, set breakpoints, and catch exceptions. Using this debugger we were able to find several nearby dynamic function pointers to overwrite as well as appropriate contents for the intervening state variables.

We modified a WMA audio file such that, when burned onto a CD, plays perfectly on a PC but sends arbitrary CAN packets of our choosing when played by our car's media player. This functionality adds only a small space overhead to the WMA file. One can easily imagine many scenarios where such an audio file might find its way into a user's media collection, such as being spread through peer-to-peer networks.

(2) OBD-II

The OBD-II port can access all CAN buses in the vehicle. This is standard functionality because the OBD-II port is the principal means by which service technicians diagnose and update individual ECUs in a vehicle. This process is intermediated by hardware tools (sold both by automobile manufacturers and third parties) that plug into the OBD-II port and can then be used to upgrade ECUs' firmware or to perform a myriad of diagnostic tasks such as checking the diagnostic trouble codes (DTCs).

Since 2004, the Environmental Protection Agency has mandated that all new cars in the U.S. support the SAE J2534 "PassThru" standard—a Windows API that provides a standard, programmatic interface to communicate with a car's internal buses. This is typically implemented as a Windows DLL that communicates over a wired or wireless network with the reprogramming/diagnostic tool (hereafter we refer to the latter simply as "the PassThru device"). The PassThru device itself plugs into the OBD-II port in the car and from that vantage point can communicate on the vehicle's internal networks under the direction of software commands sent via the J2534 API. In this way, applications developed independently of the particular PassThru device can be used for reprogramming or diagnostics.

We studied the most commonly used PassThru device for our car, manufactured by a well-known automotive electronics supplier on an OEM basis (the same device can be used for all current makes and models from the same automobile manufacturer). The device itself is roughly the size of a paperback book and consists of a popular SoC microprocessor running a variant of Linux as well as multiple network interfaces, including USB and WiFi—and a connector for plugging into the car's OBD-II port.[15] We discovered two classes of vulnerabilities with this device. First, we find that an attacker on the same WiFi network as the PassThru device can easily connect to it and, if the PassThru device is also connected to a car, obtain control over the car's reprogramming. Second, we find it possible to compromise the PassThru device itself, implant malicious code, and thereby affect a far greater number of vehicles. To be clear, these are vulnerabilities in the PassThru device itself, not the Windows software which normally communicates with it. We experimentally evaluated both vulnerability classes and elaborate on our analyses below.

[15] The manufacturer's dealership guidelines recommend the use of the WiFi interface, thereby supporting an easier tetherless mode of use, and suggest the use of link-layer protection such as WEP (or, in the latest release of the device, WPA2) to prevent outside access.

After booting up, the device periodically advertises its presence by sending a UDP multicast packet on each network to which it is connected, communicating both its IP address and a TCP port for receiving client requests. Client applications using the PassThru DLL connect to the advertised port and can then configure the PassThru device or command it to begin communicating with the vehicle. Communication between the client application and the PassThru device is unauthenticated and thus depends exclusively on external network security for any access control. Indeed, in its recommended mode of deployment, any PassThru device should be directly accessible by any dealership computer. A limitation is that only a single application can communicate with a given PassThru device at a time, and thus the attacker must wait for the device to be connected but not in use.

The PassThru device exports a proprietary, unauthenticated API for configuring its network state (e.g., for setting with which WiFi SSID it should associate). We identified input validation bugs in the implementation of this protocol that allow an attacker to run arbitrary Bourne Shell commands via shell-injection, thus compromising the unit. The underlying Linux distribution includes programs such as `telnetd`, `ftp`, and `nc`, so having gained entry to the device via shell injection, it is trivial for the attacker to open access for inbound telnet connections (exacerbated by a poor choice of root password) and then transfer additional data or code as necessary.

To evaluate the utility of this vulnerability and make it concrete, we built a program that combines all of these steps. It contacts any PassThru devices being advertised (e.g., via their WiFi connectivity or if connected directly via Ethernet), exploits them via shell injection, and installs a malicious binary (modifying startup scripts so it is always enabled). The malicious binary will send pre-programmed messages over the CAN bus whenever a technician connects the PassThru device to a car. These

CAN packets install malware onto the car's telematics unit. This malware waits for an environmental trigger (e.g., specific date and time) before performing some action. Figure 9 gives a pictorial overview of this attack.

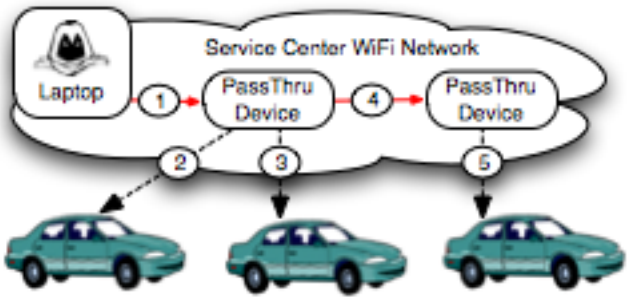

Figure 9: *PassThru-based shell-injection exploit scenario.* The adversary gains access to the service center network (e.g., by compromising an employee laptop), then (1) compromises any PassThru devices on the network, each of which compromise any cars they are used to service (2 and 3), installing Trojan horses to be activated based on some environmental trigger. The PassThru device also (4) spreads virally to other PassThru devices (e.g., if a device is loaned to other shops) which can repeat the same process (5).

To summarize, an attacker who can connect to a dealership's wireless network (e.g., via social engineering or a worm/virus a la Stuxnet [**89**]) is able to subvert any active PassThru devices that will in turn compromise any vehicles to which they connect. Moreover, the PassThru device is sufficiently general to mount the attack *itself*. To demonstrate this, we have modified our program, turning it into a worm that actively seeks out and spreads to other PassThru devices in range. This attack does not require interactivity with the attacker and can be fully automated.

3.3.3. Short-Range Wireless Channels: Bluetooth

We now turn to short-range wireless channels and focus on one in particular: Bluetooth. Like many modern cars, ours has built-in Bluetooth capabilities which allow the occupants' cell phones to connect to the car (e.g., to enable hands-free calling). These Bluetooth capabilities are built into our car's telematics unit.

Through reverse engineering, we gained access to the telematics ECU's UNIX-like operating system and identified the particular program responsible for handling Bluetooth functionality. By analyzing the program's symbols we established that it contains a copy of a popular embedded implementation of the Bluetooth protocol stack and a sample hands-free application. However, the interface to this program and the rest of the telematics system appear to be custom-built. It is in this custom interface code that we found evidence of likely vulnerabilities. Specifically, we observed over 20 calls to `strcpy`, none of which were clearly safe. We investigated the first such instance in depth and discovered an easily exploitable unchecked `strcpy` to the stack when handling a Bluetooth configuration command.[16] Thus, any paired Bluetooth device can exploit this vulnerability to execute arbitrary code on the telematics unit.

As with our indirect physical channel investigations, we establish the utility of this vulnerability by making it concrete. We explore two practical methods for exploiting this attack and in doing so unearth two sub-classes of the short-range wireless attack vector: *indirect* short-range wireless attacks and *direct* short-range wireless attacks.

(1) Indirect Short-Range Wireless Attacks

The vulnerability we identified requires the attacker to have a *paired* Bluetooth device. It may be challenging for an attacker to pair her own device with the car's Bluetooth system—a challenge we consider in the direct short-range wireless attacks discussion below. However, the car's Bluetooth subsystem was explicitly designed to support hands-free calling and thus may naturally be paired with one or more smartphones. We conjecture that if an attacker can independently compromise

[16] Because the size of the available buffer is small, our exploit simply creates a new shell on the telematics unit from which it downloads and executes more complex code from the Internet via the unit's built-in 3G data capabilities.

one of those smartphones, then the attacker can leverage the smartphone as a stepping-stone for compromising the car's telematics unit, and thus all the critical ECUs on the car.

To assess this attack vector we implemented a simple Trojan Horse application on the HTC Dream (G1) phone running Android 2.1. The application appears to be innocuous but under the hood monitors for new Bluetooth connections, checks to see if the other party is a telematics unit (our unit identifies itself by the car manufacturer name), and if so sends our attack payload. While we have not attempted to upload our code to the Android Market, there is evidence that other Trojan applications have been successfully uploaded [**90**]. Additionally, there are known exploits that can compromise Android and iPhone devices that visit malicious Web sites. Thus our assessment suggests that smartphones can be a viable path for exploiting a car's short-range wireless Bluetooth vulnerabilities.

(2) Direct Short-Range Wireless Attacks

We next assess whether an attacker can remotely exploit the Bluetooth vulnerability without access to a paired device. Our experimental analyses found that a determined attacker can do so, albeit in exchange for a significant effort in development time and an extended period of proximity to the vehicle.

There are two steps precipitating a successful attack. First, the attacker must learn the car's Bluetooth MAC address. Second, the attacker must surreptitiously pair his or her own device with the car. Experimentally, we find that we can use the open source Bluesniff [**91**] package and a USRP-based software radio to sniff our car's Bluetooth MAC address when the car is started in the presence of a previously paired device (e.g., when the driver turns on the car while carrying her cell

phone). We were also able to discover the car's Bluetooth MAC address by sniffing the Bluetooth traffic generated when one of the devices, which has previously been paired to a car, has its Bluetooth unit enabled, regardless of the presence of the car—all of the devices we experimented with scanned for paired devices upon Bluetooth initialization.

Given the MAC address, the other requirement for pairing is possessing a shared secret (the PIN). Under normal use, if the driver wishes to pair a new device, she puts the car into pairing mode via a well-documented user interface, and, in turn, the car provides a random PIN (regenerated each time the car starts or when the driver initiates the normal pairing mode) which is then shown on the dashboard and must then be manually entered into the phone. However, we have discovered that our car's Bluetooth unit will respond to pairing requests even without any user interaction. Using a simple laptop to issue pairing requests, we are thus able to brute force this PIN at a rate of eight to nine PINs per minute, for an average of approximately 10 hours per car; this rate is limited entirely by the response time of the vehicle's Bluetooth stack. We conducted three empirical trials against our car (resetting the car each time to ensure that a new PIN was generated) and found that we could pair with the car after approximately 13.5, 12.5, and 0.25 hours, respectively. The pairing process does not require any driver intervention and will happen completely obliviously to any person in the car.[17] While this attack is time consuming and requires the car(s) under attack to be running, it is also parallelizable, e.g., an attacker could sniff the MAC addresses of all cars started in a parking garage at the end of a day (assuming the cars are pre-paired with at least one Bluetooth device). If a thousand such cars leave the parking garage in a day, then we expect to be able to brute force the PIN for at least one car within a minute.

[17] As an artifact of how this "blind" pairing works, the paired device does not appear on the driver's list of paired devices and cannot be unpaired manually.

After completing this pairing, the attacker can inject on the paired channel an exploit like the one we developed and thus compromise the vehicle.

3.3.4. Long-Range Wireless Channels: Cellular

Finally, we consider long-range wireless channels and, in particular, focus on the cellular capabilities built into our car's telematics unit. Like many modern cars, our car's cellular capabilities facilitate a variety of safety and convenience features (e.g., the car can automatically call for help if it detects a crash). However, long-range communications channels also offer an obvious target for potential attackers, which we explore here. In this section, we describe how these channels operate, how they were reverse engineered and demonstrate that a combination of software flaws conspire to allow a completely remote compromise via the cellular voice channel. We focus on adversarial actions that leverage the existing cellular infrastructure, not ones that involve the use of adversarially-controlled infrastructure; e.g., we do not consider man-in-the-middle attacks.

(1) Telematics Connectivity

For wide-area connectivity, our telematics unit is equipped with a cell phone interface (supporting voice, SMS and 3G data). While the unit uses its 3G data channel for a variety of Internet-based functions (e.g., navigation and location-based services), it relies on the voice channel for critical telematics functions (e.g., crash notification) because this medium can provide connectivity over the widest possible service area (i.e., including areas where 3G service is not yet available). To synthesize a digital channel in this environment, the manufacturer uses Airbiquity's aqLink software modem to covert between analog waveforms and digital bits. This use of the voice channel in general, and the aqLink software in particular, is common to virtually all popular North American telematics offerings today.

In our vehicle, Airbiquity's software is used to create a reliable data connection between the car's telematics unit and a remote *Telematics Call Center* (*TCC*) operated by the manufacturer. In particular, the telematics unit incorporates the aqLink code in its *Gateway* program which controls *both* voice and data cellular communication. Since a single cellular channel is used for both voice and data, a simple, in-band, tone-based signaling protocol is used to switch the call into data mode. The in-cabin audio is muted when data is transmitted, although a tell-tale light and audio announcement is used to indicate that a call is in progress. For pure data calls (e.g., telemetry and remote diagnostics), the unit employs a so-called "stealth" mode which does not provide any indication that a call is in progress.

(2) Reverse Engineering the aqLink Protocol

Reverse engineering the aqLink protocol was among the most demanding parts of our effort, in particular because it demanded signal processing skills not part of the typical reverse engineering repertoire. For pedagogical reasons, we briefly highlight the process of our investigation.

We first identified an in-band tone used to initiate "data mode." Having switched to data mode, aqLink provides a proprietary modulation scheme for encoding bits. By calling our car's telematics unit (the phone number is available via caller ID), initiating data mode with a tone generator and recording the audio signal that resulted, we established that the center frequency was roughly 700 Hz and that the signal was consistent with a 400 bps frequency-shift keying (FSK) signal.

We then used `LD_PRELOAD` on the telematics unit to interpose on the raw audio samples as they left the software modem. Using this improved signal source, we hunted for known values contained in the signal (e.g., unique identifiers stamped on the unit). We did so by encoding these values as binary waveforms at hypothesized bitrates and cross-correlating them to the demodulated signal

until we were able to establish the correct parameters for demodulating digital bits from the raw analog signal.

From individual bits, we then focused on packet structure. We were lucky to discover a debugging flag in the telematics software that would produce a binary log of all packet payloads transmitted or received, providing ground truth. Comparing this with the bitstream data, we discovered the details of the framing protocol (e.g., the use of half-width bits in the synchronization header) and were able to infer that data is sent in packets of up to 1024-bytes, divided into 22-byte frames which are divided into two 11-byte segments. We inferred that a CRC and ECC were both used to tolerate noise. Searching the disassembled code for known CRC constants quickly led us to determine the correct CRC to use, and the ECC code was identified in a similar fashion. For reverse-engineering the header contents, we interposed on the `aqSend` call (used to transmit messages), which allowed us to send arbitrary multi-frame packets and consequently infer the sequence number, multi-frame identifier, start of packet bit, ACK frame structure, etc.

Given our derived protocol specification, we then implemented an aqLink-compatible software modem in C using a laptop with an Intel ICH3-based modem exposed as an ALSA sound device under Linux. We verified the modulation and formatting of our packet stream using the debugging log described earlier.

Finally, layered on top of the aqLink modem is the telematics unit's own proprietary command protocol that allows the TCC to retrieve information about the state of the car as well as to remotely actuate car functions. Once the Gateway program decodes a frame and identifies it as a command message, the data is then passed (via an RPC-like protocol) to another telematics unit program

which is responsible for supervising overall telematics activities and implementing the command protocol (henceforth, the *Command* program). We reverse-engineered enough of the Gateway and Command programs to identify a candidate vulnerability, which we describe below.

The custom code that glues aqLink to the Command program assumes that packets will never exceed 100 bytes or so (presumably since well-formatted command messages are always smaller). This leads to another stack-based buffer overflow vulnerability that we verified is exploitable. Interestingly, because this attack takes place at the lowest level of the protocol stack, it completely bypasses the higher-level authentication checks implemented by the Command program (since these checks themselves depend on being able to send packets).

There is one key gap preventing this exploit from working in practice. Namely, the buffer overflow we chose to focus on requires sending over 300 bytes to the Gateway program. Since the aqLink protocol has a maximum effective throughput of about 21 bytes a second, in the best case, the attack requires about 14 seconds to transmit. However, upon receiving a call, the Command program sends the caller an authentication request and, serendipitously, it requires a response within 12 seconds or the connection is effectively terminated. Thus, we simply cannot send data fast enough over an unauthenticated link to overflow the vulnerable buffer.

While we identified other candidate buffer overflows of slightly shorter length, we decided instead to focus on the authentication problem directly.

(3) Vulnerabilities in Authentication

When a call is placed to the car and data mode is initiated, the first command message sent by the vehicle is a random, three byte authentication challenge packet and the Command program authentication timer is started. In normal operation, the TCC hashes the challenge along with a 64-bit pre-shared key to generate a response to the challenge. When waiting for an authentication response, the Command program will not "accept" any other packet (this does not prevent our buffer overflow, but does prevent sending other command messages). If an incorrect authentication response is received, or a response is not received within the prescribed time limit, the Command program will send an error packet. When this packet is acknowledged, the unit hangs up (and it is not possible to send any additional data until the error packet is acknowledged).

After several failed attempts to derive the shared key, we examined code that generates authentication challenges and evaluates responses. Both contained errors that together were sufficient to construct a vulnerability.

First, we noted that the "random" challenge implementation is flawed. In most situations, this nonce is static and identical on the two cars we tested. The key flaw is that the random number generator is re-initialized whenever the telematics unit starts—such as when a call comes in after the car has been off—and it is seeded each time with the same constant. Therefore, multiple calls to a car while it is off result in the same expected response. Consequently, an attacker able to observe a response packet (e.g., via sniffing the cellular link during a TCC-initiated call) will be able to replay that response in the future.

The code parsing authentication *responses* has an even more egregious bug that permits circumvention without observing a correct response. In particular, there is a flaw such that for certain challenges (roughly one out of every 256), carefully formatted but incorrect responses will be interpreted as valid. If the random number generation is not re-initialized (e.g., if the car is on when repeatedly called) then the challenge will change each time and 1 out of 256 trials will have the desired structure. Thus, after an average of 128 calls the authentication test can be bypassed, and we are able to transmit the exploit (again, without any indication to the driver). This attack is more challenging to accomplish when the car is turned off because the telematics unit can shut down when a call ends (hence re-initializing the random number generator) before a second call can reach it.

To summarize, we identified several vulnerabilities in how our telematics unit uses the aqLink code that, together, allow a remote exploit. Specifically, there is a discrepancy between the set of packet sizes supported by the aqLink software and the buffer allocated by the telematics client code. However, to exploit this vulnerability requires first authenticating in order to set the call timeout value long enough to deliver a sufficiently long payload. This is possible due to a logic flaw in the unit's authentication system that allows an attacker to blindly satisfy the authentication challenge after approximately 128 calls.

(4) Concrete Realization

We demonstrate and evaluate our attack in two concrete forms. First, we implemented an end-to-end attack in which a laptop running our custom aqLink-compatible software modem calls our car repeatedly until it authenticates, changes the timeout from 12 seconds to 60 seconds, and then re-calls our car and exploits the buffer overflow vulnerability we uncovered. The exploit then forces

the telematics unit to download and execute additional payload code from the Internet using the IP-addressable 3G data capability.

We also found that the entire attack can be implemented in a completely blind fashion—without any capacity to listen to the car's responses. Demonstrating this, we encoded an audio file with the modulated post-authentication exploit payload and loaded that file onto an iPod. By manually dialing our car on an office phone and then playing this "song" into the phone's microphone, we are able to achieve the same results and compromise the car.

3.4. Remote Exploit Control

Thus far we have described the external attack surface of an automobile and demonstrated the presence of vulnerabilities in a range of different external channels. An adversary could use such means to compromise a vehicle's systems and install code that takes action immediately (e.g., unlocking doors) or in response to some environmental trigger (e.g., the time of day, speed, or location as exported via the onboard GPS).

However, the presence of wireless channels in the modern vehicle qualitatively changes the range of options available to the adversary, allowing actions to be remotely triggered on demand, synchronized across multiple vehicles, or interactively controlled. Further, two-way channels permit both remote monitoring and data exfiltration. In this section, we broadly evaluate the potential for such post-compromise control, characterize these capabilities, and evaluate the capabilities via prototype implementations for TPMS, Bluetooth, FM RDS and Cellular channels. Our prototype attack code is delivered by exploiting one of the previously described vulnerabilities (indeed, any

exploit would work). Table 7 summarizes these results, again with our assessment of the effort required to discover and implement the capability.

Channel	Range	Implemented Control / Trigger	Exfiltration	Cost
TPMS (tire pressure)	Short	Predefined tire pressure sequences causes telematics unit to send CAN packets	No	Low-Medium
TPM (tire pressure)	Short	TPMS trigger causes TPMS receiver to send CAN packets	No	Medium
Bluetooth	Short	Presence of trigger MAC addresses allow remote control	Yes*	Low
FM radio (RDS)	Long	FM RDS trigger causes radio to send CAN packets	No	Medium
Cellular	Global	IRC command-and-control (botnet) channel allows broadcast and single-vehicle command	Yes	Low

Table 7: *Implemented control and trigger channels.* The Cost column captures the approximate effort to develop this post-compromise control capability. The Exfiltration column indicates whether this channel can also be used to exfiltrate data. For (*), we did not experimentally verify data exfiltration over Bluetooth.

3.4.1. TPMS

We constructed two versions of a TPMS-based triggering channel. One installs code on another ECU (the telematics ECU in our case, although any ECU would do) that monitors tire pressure signals as the TPMS ECU broadcasts them over the CAN bus. The presence of a particular tire pressure reading then triggers the payload; the trigger tire pressure value is not expected to be found in the wild but must instead be adversarially transmitted over the air. For our second example, the

attack reflashes the TPMS ECU via CAN and installs code onto it that will detect specific wireless trigger packets and, if detected, will send pre-programmed CAN packets directly over the car's internal network. Both attacks required a custom TPMS packet generator (described below). The latter attack also required significant reverse engineering efforts (e.g., we had to write a custom IDA Pro module for disassembling the firmware, and we were highly memory constrained, so that the resulting attack firmware—hand-written object code—needed to re-use code space originally allocated for CRC verification, the removal of which did not impair the normal TPMS functionality).

To experimentally verify these triggers, we reverse-engineered the 315 MHz TPMS modulation and framing protocol (far simpler than the aqLink modem) and then implemented a USRP software radio module that generates the appropriate wireless signals to activate the triggers.

3.4.2. Bluetooth

We modified the Bluetooth exploit code on the telematics ECU to pair, post compromise, with a special MAC address used by the adversary and accept her commands (either triggering existing functionality or receiving new functionality). We did not explore exfiltrating data via the two-way Bluetooth channel, but we see no reason why it would not be possible.

3.4.3. FM RDS

Using the CD-based firmware update attack we developed earlier, we reflashed the media player ECU to send a pre-determined set of CAN packets (our payload) when a particular "Program Service Name" message arrives over the FM RDS channel. We experimentally verified this with a low-power FM transmitter driven by a Pira32 RDS encoder; an attacker could communicate over much longer ranges using higher power. Table 7 lists the cost for this attack as medium given the

complexity of programming/debugging in the media player execution environment (we bricked numerous CD players before finalizing our implementation and testing on our car).

3.4.4. Cellular

We modified our telematics exploit payload to download and run a small (400 lines of C code) IRC client post-compromise. The IRC client uses the vehicle's high bandwidth 3G data channel to connect to an IRC server of our choosing, self-identifies, and then listens for commands. Subsequently, any commands sent to this IRC server (from any Internet connected host) are in turn transmitted to the vehicle, parsed by the IRC client, and then transmitted as CAN packets over the appropriate bus. We further provided functionality to use this channel in both a broadcast mode (where all vehicles subscribed to the channel respond to the commands) or selectively (where commands are only accepted by the particular vehicle specified in the command). For the former, we experimentally verified this by compromising two cars (located over 1,000 miles apart), having them both join the IRC channel, and then both simultaneously respond to a single command (for safety, the command we sent simply made the audio systems on both cars chime). Finally, the high-bandwidth nature (up to 1 Mbps at times) of this channel makes it easy to exfiltrate data. (No special software is needed since `ftp` is provided on the host platform.) To make this concrete we modified our attack code for two demonstrations: one that periodically "tweets" the GPS location of our vehicle and another that records cabin audio conversations and sends the recorded data to our servers over the Internet.

3.5. Threat Assessment

Thus far we have considered threats primarily at a technical level. In Chapter 2, we demonstrated that gaining access to a car's internal network provides sufficient *means* for compromising all of its

systems (including lights, brakes, and engine). In this chapter, we have further demonstrated that an adversary has a practical *opportunity* to affect this compromise (i.e., via a range of external communications channels) without having physical access to the vehicle. However, real threats ultimately have some *motive* as well: a more concrete goal that is achieved by exploiting the capability to attack.

This leaves unanswered the crucial question: Just how serious are the threats? Obviously, there are no clear ways to predict such things, especially in the absence of any known attacks in the wild. However, we can reason about how the capabilities we have identified can be combined in service to known goals. While one can easily envision hypothetical "cyber war" or terrorist scenarios (e.g., infect large numbers of cars en masse via war dialing or a popular audio file and then, later, trigger them to simultaneously disengage the brakes when driving at high speed), our lack of experience with such concerns means such threats are highly speculative.

Instead, to gauge whether these threats create practical risks, we consider (briefly) how the raw capabilities we have identified might affect two scenarios closer to our experience: financially motivated theft and third-party surveillance.

3.5.1. Theft

Using any of our implemented exploit capabilities (CD, PassThru, Bluetooth, and cellular), it is simple to command a car to unlock its doors on demand, thus enabling theft. However, a more visionary car thief might realize that blind, remote compromise can be used to change both scale and, ultimately, business model. For example, instead of attacking a *particular* target car, the thief might instead try to compromise as many cars as possible (e.g., by war dialing). As part of this

compromise, he might command each car to contact a central server and report back its GPS coordinates and Vehicle Identification Number (VIN). The IRC network described in Section 3.4.4 provides just this capability. The VIN in turn encodes the year, make and model of each car and hence its value. Putting these capabilities together, a car thief could "sift" through the set of cars, identify the valuable ones, find their location (and perhaps how long they have been parked) and, upon visiting a target of interest *then* issue commands to unlock the doors and so on. An enterprising thief might stop stealing cars himself, and instead sell his capabilities as a "service" to other thieves ("I'm looking for late model BMWs or Audis within a half mile of 4th and Broadway. Do you have anything for me?") Careful readers may notice that this progression mirrors the evolution of desktop computer compromises: from individual attacks, to mass exploitation via worms and viruses, to third-party markets selling compromised hosts as a service.

While the scenario itself is today hypothetical, we have evaluated a complete attack whereby a thief remotely disables a car's security measures, allowing an unskilled accomplice to enter the car and drive it away. Our attack directs the car's compromised telematics unit to unlock the doors, start the engine, disengage the shift lock solenoid (which normally prevents the car from shifting out of park without the key present), and spoof packets used in the car's startup protocol (thereby bypassing the existing immobilizer anti-theft measures[18]). We have implemented this attack on our car. In our experiments the accomplice only drove the "stolen" car forward and backward because we did not want to break the steering column lock, though numerous online videos demonstrate how to do so using a screwdriver. (Other vehicles have the steering column lock under computer control.)

[18] Past work on bypassing immobilizers required prior direct or indirect access to the car's keys, e.g., Bono et al. [**29**] and Francillon et al. [**28**]

3.5.2. Surveillance

We have found that an attacker who has compromised our car's telematics unit can record data from the in-cabin microphone (normally reserved for hands-free calling) and exfiltrate that data over the connected IRC channel. Moreover, as said before, it is easy to capture the location of the car at all times and hence track where the driver goes. These capabilities, which we have experimentally evaluated, could prove useful to private investigators, corporate spies, paparazzi, and others seeking to eavesdrop on the private conversations within particular vehicles. Moreover, if the target vehicle is not known, the mass compromise techniques described in the theft scenario can also be brought to bear on this problem. For example, someone wishing to eavesdrop on Google executives might filter a set of compromised cars down to those that are both expensive and located in the Google parking lot at 10 a.m. The location of those same cars at 7 p.m. is likely to be the driver's residence, allowing the attacker to identify the driver (e.g., via commercial credit records). We suspect that one could identify promising targets for eavesdropping quite quickly in this manner.

3.6. Discussions and Synthesis

Our research provides us with new insights into the risks with modern automotive computing systems. We begin here with a discussion of concrete directions for increasing security. We then turn to our now broadly informed reflections on why vulnerabilities exist today and the challenges in mitigating them.

3.6.1. Implementation Fixes

Our concrete, near-term recommendations fall into two familiar categories: restrict access and improve code robustness. Given the high interconnectedness of car ECUs necessary for desired

functionality, the solution is not to simply remove or harden individual components (e.g., the telematics unit) or create physically isolated sub-networks.

We were surprised at the extent to which the car's externally facing interfaces were open to unsolicited communications—thereby broadening the attack surface significantly. Indeed, very simple actions, such as not allowing Bluetooth pairing attempts without the driver's first manually placing the vehicle in pairing mode, would have undermined our ability to exploit the vulnerability in the underlying Bluetooth code. Similarly, we believe the cellular interface could be significantly hardened by using inbound calls only to "wake up" the car (i.e., never for data transfer) and having the car itself periodically dial out for requests while it is active. Finally, use of application-level authentication and encryption (e.g., via OpenSSL) in the PassThru device's proprietary configuration protocol would have prevented its code from being exploited as well.

However, rather than assume the attack surface will not be breached, the underlying code platform should be hardened as well. These include standard security engineering best-practices, such as not using unsafe functions like `strcpy`, diligent input validation, and checking function "contracts" at module boundaries. As an additional measure of protection against less-motivated adversaries, we recommend removing all debugging symbols and error strings from deployed ECU code.

We also encourage the use of simple anti-exploitation mitigations such as stack cookies and ASLR that can be easily implemented even for simple processors and can significantly increase the exploit burden for potential attackers. In the same vein, critical communications channels (e.g., Bluetooth and telematics) should have some amount of behavioral monitoring. The car should not allow arbitrary numbers of connection failures to go unanswered nor should outbound Internet

connections to arbitrary destinations be allowed. In cases where ECUs communicate on multiple buses, they should only be allowed to be reflashed from the bus with the smallest external attack surface. This does not stop all attacks where one compromised ECU affects an ECU on a bus with a smaller attack surface, but it does make such attacks more difficult. 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.

Finally, secure (authenticated and reliable) software updates must also be considered as part of automotive component design.

3.6.2. Vulnerability Drivers

While the recommendations in Section 3.6.1 can significantly increase the security of modern cars against external attacks and post-compromise control, none of these ideas are new or innovative. Thus, perhaps the more interesting question is why they have not been applied in the automotive environment already. Our findings and subsequent interactions with the automotive industry have given us a unique vantage point for answering this question.

One clear reason is that automobiles have not yet been subjected to significant adversarial pressures. Traditionally automobiles have not been network-connected and thus manufacturers have not had to anticipate the actions of an external adversary; anyone who could get close enough to a car to modify its systems was also close enough to do significant damage through physical means. Our

automotive systems now have broad connectivity; millions of cars on the road today can be directly addressed via cellular phones and via the Internet.

This is similar to the evolution of desktop personal computer security during the early 1990s. In the same way that connecting PCs to the Internet exposed extant vulnerabilities that previously could not conveniently be exploited, so too does increasing the connectivity of automotive systems. This analogy suggests that, even though automotive attacks do not take place today, there is cause to take their potential seriously. Indeed, much of our work is motivated by a desire that the automotive manufacturers should not repeat the mistakes of the PC industry—waiting for high profile attacks before making security a top priority [**2**] [**3**]. We believe many of the lessons learned in hardening desktop systems (such as those suggested earlier) can be quickly re-purposed for the embedded context.

However, our experimental vulnerability analyses also uncover an ecosystem for which high levels of assurance may be fundamentally challenging. Reflecting upon our discovered vulnerabilities, we noticed interesting similarities in where they occur. In particular, virtually all vulnerabilities emerged at the interface boundaries between code written by distinct organizations.

Consider for example the Airbiquity software modem, which appears to have been delivered as a completed component. We found vulnerabilities not in the software modem *itself* but rather in the "glue" code calling it and binding it to other telematics functions. It was here that the caller did not appear to fully understand the assumptions made by the component being called.

We find this pattern repeatedly. The Bluetooth vulnerability arose from a similar misunderstanding between the callers of the Bluetooth protocol stack library and its implementers (again in glue code). The PassThru vulnerability arose in script-based glue code that tried to interface a proprietary configuration protocol with standard Linux configuration scripts. Even the media player firmware update vulnerability appears to have arisen because the manufacturer was unaware of the vestigial CD-based reflashing capability implemented in the code base.

While interface boundary problems are common in all kinds of software, we believe there are structural reasons that make them particularly likely in the automotive industry. In particular, the automotive industry has adopted an outsourcing approach to software that is quite similar to that used for mechanical components: supply a specification and contract for completed parts. Thus, for many components the manufacturer does not do the software development and is only responsible for integration. We have found, for example, that different model years of ECUs with effectively the same functionality used completely different source code bases because they were provided by different suppliers. Indeed, we have come to understand that frequently manufacturers do not have access to the source code for the ECUs they contract for (and suppliers are hesitant to provide such code since this represents their key intellectual property advantage over the manufacturer). Thus, while each supplier does unit testing (according to the specification) it is difficult for the manufacturer to evaluate security vulnerabilities that emerge at the integration stage. Traditional kinds of automated analysis and code reviews cannot be applied and assumptions not embodied in the specifications are difficult to unravel. Therefore, 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.

Developing security solutions compatible with the automotive ecosystem is challenging and we believe it will require more engagement between the computer security community and automotive manufacturers (in the same way that our community engages directly with the makers of PC software today).

3.7. Conclusions

A modern automobile is controlled by tens of distinct computers physically interconnected with each other via internal (wired) buses and thus exposed to one another. A non-trivial number of these components are also externally accessible via a variety of I/O interfaces. Previous research showed that an adversary can seriously impact the safety of a vehicle if he or she is capable of sending packets on the car's internal wired network [**87**], and numerous other papers have discussed potential security risks with future (wired and wireless) automobiles in the abstract or on the bench [**26**] [**21**] [**22**] [**23**] [**24**] [**25**]. To the best of our knowledge, however, we are the first to experimentally and systematically study the externally-facing attack surface of a car.

Our experimental analyses focus on a representative, moderately priced sedan. We iteratively refined an automotive threat model framework and implemented complete, end-to-end attacks along key points of this framework. For example, we can compromise the car's radio and upload custom firmware via a doctored CD, we can compromise the technicians' PassThru devices and thereby compromise any car subsequently connected to the PassThru device, and we can call our car's cellular phone number to obtain full control over the car's telematics unit over an arbitrary distance. Being able to compromise a car's ECU is, however, only half the story: The remaining concern is what an attacker is able to do with those capabilities. In fact, we show that a car's externally-facing I/O interfaces can be used post-compromise to remotely trigger or control arbitrary vehicular

functions at a distance and to exfiltrate data such as vehicle location and cabin audio. Finally, we consider concrete, financially-motivated scenarios under which an attacker might leverage the capabilities we develop in this chapter.

Our experimental results give us the unique opportunity to reflect on the security and privacy risks with modern automobiles. We synthesize concrete, pragmatic recommendations for future automotive security, as well as identify fundamental challenges. We disclosed our results to relevant industry and government stakeholders. While defending against known vulnerabilities does not imply the non-existence of other vulnerabilities, many of the specific vulnerabilities identified in this work have or will soon be addressed.

Chapter 4: SURROGATES: Enabling Near-Real-Time Dynamic Analysis of Embedded Systems

4.1. Introduction

Embedded systems are becoming increasingly sophisticated, inter-connected, and pervasive, making the "Internet of Things" the hot new buzzword. Unfortunately, as demonstrated in the past few chapters, these systems have been repeatedly shown to be insecure. Even if manufacturers want to build secure products, the security tools available to embedded systems developers pale in comparison to those for traditional software.

In particular, dynamic analysis techniques are challenging to apply due to the difficulty of instrumenting embedded systems. There may not be sufficient storage space for an instrumented binary or its measurements. There may not be sufficient processing power for instrumentation. There may not be a way to provide arbitrary data to the system—a necessity for fuzzing. Even if a system is technically capable of added instrumentation, firmware heterogeneity requires substantial work to customize instrumentation for each device. Whereas traditional software runs on top of a few standard OSes (with standard facilities that support instrumentation, such as a file system and dynamic linker), embedded systems may not even have an OS! The analyst must identify instrumentation points and storage available for measurements, and surgically insert code into the firmware.

An alternative to placing instrumentation on the device itself is to run the system under emulation. However, this introduces its own set of challenges. Embedded systems are highly intertwined with their environment, through sensors, actuators, and other interfaces. Furthermore, the peripherals that control these interfaces can vary a great deal from one device to another. Faithfully emulating these peripherals again requires a great deal of work building customized solutions.

One approach to this problem, as described in Section 1.3.4, is to treat peripherals as unconstrained symbolic inputs. However, this relies on the analysis using symbolic execution. Unconstrained inputs can lead to state explosion, rendering this technique unsuitable for all but the smallest embedded systems.

We take a different approach. Like Avatar [**82**] (also described in Section 1.3.4), we run the device's firmware under emulation, directing peripheral I/O to the actual device, giving the emulated firmware a realistic view of its environment. This leverages the fact that many devices rely on a relatively small set of embedded processors; System-on-Chip (SoC) manufacturers typically license a well-known CPU core and add their own custom peripherals. Symbolic analysis techniques can further leverage concrete knowledge of system behavior to constrain the explored state space using a technique called *concolic* (a portmanteau of "concrete" and "symbolic") execution.

However, there are a number of challenges in making this approach work without being prohibitively slow. Avatar attempts to overcome these challenges by limiting the amount firmware executed under emulation. However, this raises a number of additional problems. The analyst must have sufficient insight into operation of the firmware to decide which parts are interesting enough to run under emulation. Emulated code still executes slowly, so this technique may not work with timing-sensitive devices (such as a medical device with a watchdog coprocessor.) Even when it does work, it does not provide a feasible way to do whole-system analysis.

Instead of limiting the scope of emulated execution, we introduce a system called SURROGATES, which can emulate *entire* systems in *near real-time*. We accomplish this by using custom, low-latency hardware to bridge the PCI Express bus of the host to the device under test, as well as making a number of optimizations. In

doing so, we uncover and surmount new challenges in emulating entire systems, such as handling interrupts, DMA, and clocking changes.

In this chapter, we make the following contributions: 1) We describe new hardware which enables near-real time emulation of arbitrary ARM-based embedded systems; 2) We discuss the engineering tradeoffs in building SURROGATES and provide comprehensive performance evaluations of the different techniques; 3) We describe and solve several issues that arise when emulating entire systems; and 4) We demonstrate the practicality of using our system on a diverse set of devices.

The rest of this chapter is organized as follows. Section 4.2 discusses a number of options to improve the performance of Avatar, guiding the design of SURROGATES, which is introduced in Section 4.3. Finally, Section 4.4 evaluates the performance of SURROGATES, compares it to other solutions, and describes our experience applying our system to a variety of embedded systems.

4.2. Towards Real-Time I/O

Our system targets ARM processors, which are ubiquitous in medium-to-high complexity embedded devices, and communicates over the JTAG interface exposed on most microcontrollers. JTAG has several nice properties: 1) it is usually present in embedded devices for programming and testing during manufacturing; 2) JTAG pins are usually dedicated for programming and debugging, so it provides a communications channel that is not already being used for some other purpose during normal operation; 3) JTAG interfaces tend to support high transfer rates (e.g., ARM processors can support JTAG clock rates up to $1/6^{th}$ of the core processor speed), limited primarily by off-chip factors such as connection length; and 4) existing JTAG tools can be used to read and write arbitrary memory addresses on a device, making it easy to rapidly develop an Avatar-like prototype.

JTAG interfaces expose a simple, standard state machine that can be driven by a JTAG adapter. This state machine lets the JTAG adapter select, capture, and update either a JTAG instruction register or a data register. These registers act like shift registers; data is shifted in and out simultaneously. While there is only one instruction register, several different data registers (called scan chains) can be selected using the different JTAG instructions.

As with Avatar, we first redirected emulated I/O to the target over JTAG using OpenOCD [**92**] (an open-source JTAG program). We initially used OpenOCD's built-in GDB protocol interface to initiate reads and writes and control the processor's state. However, memory operations are extremely slow over regular JTAG interfaces. This is because these memory operations are *injected* into the CPU's state. The JTAG interface must halt the CPU, transfer the CPU's state, update the CPU's state to perform a memory operation (including general purpose registers and the instruction register), single-step the CPU, read out the CPU's state again if the memory operation was a read, restore the CPU's original state, and resume the CPU.

While exposing the CPU's state over JTAG gives debuggers extremely powerful control over the system, its performance is poor for common tasks, such as transferring large segments of memory. To improve performance of these operations, CPU vendors have introduced additional scan chains that expose small communications channels between the JTAG interface and a program running on the CPU. For example, most ARM processors support the Debug Communications Channel (DCC), which is a 32-bit register accessible over a separate JTAG scan chain. JTAG interfaces can upload a small stub to the target and use the DCC to transfer large portions of memory efficiently.

We leveraged the relatively fast DCC by developing a custom stub that runs on the target, accepting memory read and write commands from the host. A full discussion of our stub and DCC protocol is in

Section 4.3.2. We modified QEMU [**93**] to directly pass selected reads and writes as DCC commands to a Segger J-Link, a commercial, off-the-shelf USB JTAG interface.

Unfortunately, we then encountered an unexpected bottleneck: USB transaction latency. USB requires all communications to be initiated by the host. This requires the host to periodically poll all devices for their status. The maximum polling rate is 1 kHz, which imposes a minimum latency of 1 ms on each USB transaction. While this may sound insignificant, it is several orders of magnitude slower than the latency of native I/O operations. Furthermore, because code execution may depend on the result of a memory read, this effectively places an upper-limit on the number of memory operations we can perform per second. This latency is a fundamental limitation of USB, which means that we must look at other interfaces to overcome it.

4.3. Our Approach: SURROGATES

4.3.1. The Hardware

We decided to avoid all unknown latencies that might be lurking in other interfaces (such as Ethernet and Firewire) by developing a custom JTAG adapter that connects directly to the host's PCI Express bus. Our goal was to transparently map the target's entire 32-bit physical address space into the 64-bit address space of the emulator, such that peripheral I/O is simply a memory read or write by the emulator. While we could not quite achieve this for reasons explained later, our JTAG interface is directly memory-mapped into the emulator process, giving us extremely low-latency access to the target. We still use our DCC stub to communicate with the target processor.

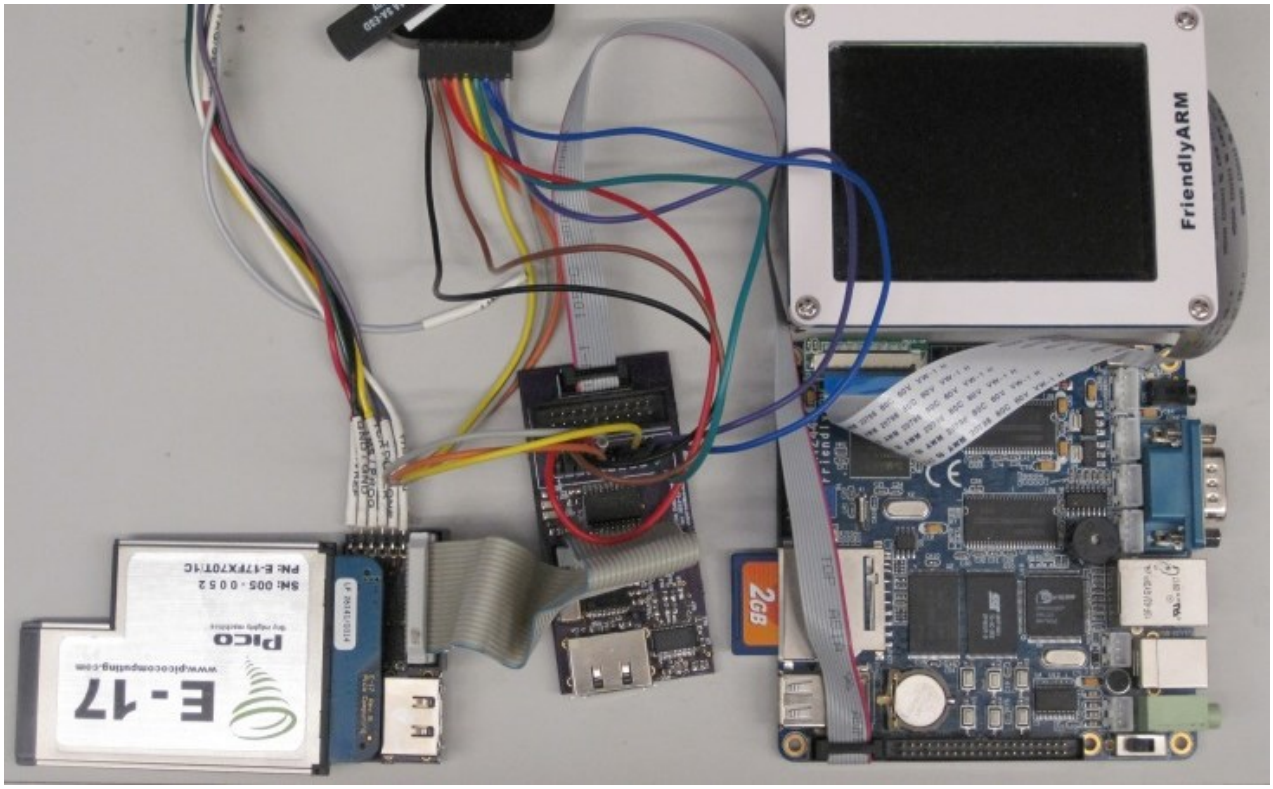

Figure 10: *Hardware components of our system*. Left-to-right: An off-the-shelf FPGA ExpressCard, our JTAG adapter board, a JTAG breakout/debug board, and the device under test (a FriendlyARM Mini2440). FPGA development and debugging is done through another JTAG connection via the JTAG interface board, as well as a small logic analyzer connected to the JTAG breakout/debug board

The PCI Express bus is not really a bus at all, but a packet-switched network. The *root complex* translates CPU reads and writes into PCI Express packets, which get routed by address. (Alternate routing schemes can be used, e.g., for device discovery and configuration.) Writes are *posted* transactions which complete immediately, while reads are *unposted*, which require a completion packet (usually with data) to be sent back to the root complex. Since PCI Express is a packet-switched network, devices can send packets to other devices, as well as perform DMA by sending packets to the root complex.

Our hardware consists of an off-the-shelf PCI Express FPGA card (a Pico Computing E17FX70T), a custom FPGA-to-JTAG interface board, and a custom JTAG debugging board, as shown in Figure 10. The

FPGA-to-JTAG board shifts signal voltage levels between the FPGA and the target's JTAG interface, and provides a standard ARM JTAG connector. It also provides a SATA-like, high-speed serial interface that can transport JTAG signals over a longer distance. The JTAG debugging board can convert this serial stream back to a standard JTAG interface, and provides an easy interface for a logic analyzer to examine the JTAG signals.

Our implementation uses a Xilinx Virtex5 FX70T FPGA. While this FPGA is overkill for our purposes, it was available off-the-shelf as a PCI Express card, with the bulk of the PCI Express glue logic already developed by Xilinx and Pico Computing. Our application logic is implemented in approximately 1,100 lines of Verilog, excluding tests (which are approximately another 1,000 lines of Verilog). Device utilization is summarized in Table 8.

	Used	*Available*	*Utilization*
Slice Registers	6,503	44,800	14%
Slice LUTs	6,615	44,800	14%
Occupied Slices	3,397	11,200	30%
BlockRAMs/FIFOs	11	148	7%
Total Memory (KB)	306	5,328	5%

Table 8: FPGA Resource Utilization

We implement two PCIe-to-JTAG bridges in the FPGA. The first is a simple set of FIFOs for the TDI, TMS, and TDO signals, and supports generic JTAG operations, such as manipulating the processor's state, dumping firmware, and uploading code. We extend OpenOCD to support this new interface and use it for some complicated-but-infrequent operations, such as resetting the target to a known state and uploading the stub.

The second interface is designed specifically to work with our stub. The original intention was to provide a transparent mapping of the target's 32-bit physical address space somewhere in the host's 64-bit address

space. Unfortunately, the PCI Express specification requires that all 64-bit address ranges be *prefetchable*—meaning that reads are side-effect free. This is not the case for several embedded devices. For example, a UART controller may have a single, memory-mapped character register. A read from this register frees the UART to receive another byte. While some chipsets do allow 64-bit PCI Express regions to not be prefetchable, others do not.

Of course, only a portion of the target's 32-bit address space is mapped to peripherals. We considered transparently mapping a small view of the target's address space, allowing the host to pick the address range that is mapped in. However, on a typical PC, there is a great deal of contention for I/O addresses below the 4 GB boundary. This makes it difficult to map reasonably large 32-bit regions. Furthermore, devices typically use large peripheral address spaces (e.g., 320 MB on the Samsung S3C2440) even though they are sparsely populated. Since the host may have to keep remapping different views of the target's address space, we decided to simply expose a few memory-mapped registers that initiate reads and writes to the target. These registers are described below and shown in Table 9.

There are two address registers—one for reads, and one for writes, as well as a data register. When a write address and value are written, the FPGA initiates a write operation on the target through its DCC interface. When an address is written to the read address register, a read operation on the target is initiated. We also provide two FIFOs and control registers to allow the host to initiate optimized multiple-word transactions.

The packet-based nature of PCI Express lets us stall reads of the data register if the target has not returned data yet. However, while the root complex is supposed to abort transactions that have timed out, our particular root complex does not. This means that if the target device does not respond (due to a bug, being

powered off, etc.), the host will freeze. Not even the NMI watchdog can recover the system. For this reason, we polled the FPGA for completion during development.

The FPGA can be configured to continuously poll the target's DCC register when there are no other pending read or write requests to check if an interrupt has occurred. Interrupts received from the stub are dispatched as interrupts to the host's processor. This required a small modification to the PCI Express interface code. The preferred way of sending interrupts over PCI Express is to use *Message Signaled Interrupts (MSIs),* which are simply memory writes of a specific value to a specific address. Peripherals no longer have to share a total of four interrupt signals (as they did with PCI), and can in fact request multiple interrupts. This would appear to allow the hardware to send different interrupts to the host based on the target's interrupt type. Unfortunately, Linux has limited support for multiple interrupts per peripheral, so the driver must poll the hardware to determine the interrupt type, as described in Section 4.3.3.

4.3.2. The Stub

Our stub targets most microcontrollers based on ARMv4T or newer cores. (Some newer ARM Cortex cores have different debugging options and capabilities.) This covers a wide range of interesting embedded devices, including hard drives, cellular baseband processors, medical devices, and automotive systems. The stub is implemented in approximately 400 lines of assembly and takes up only 768 bytes—which can be easily locked into the L1 instruction cache. The stub does not use any RAM for data or a stack, allowing the emulator to use all available RAM on the target if desired.

Our stub uses a custom word-based protocol to efficiently perform memory operations as well as transferring status information, such as interrupts and interrupt masks. A summary of our protocol is listed in Table 10.

The stub provides handlers for standard (IRQ) and fast (FIQ) interrupts. Unlike Avatar, no de-multiplexing is attempted. When an interrupt is received, ARM processors update their Current Program Status Register (CPSR) to set the IRQ or FIQ Disable bit, preventing the handler from being interrupted itself. The old CPSR value is stored in the Saved Program Status Register (SPSR). Normally when the handler returns, the SPSR is copied back to the CPSR, re-enabling interrupts. However, we adjust the SPSR to keep interrupts disabled and deliver the interrupt type to the host.

The host delivers the interrupt to the emulated processor when its CPSR is set to allow interrupts. The emulated firmware can then query the interrupt controller like any other peripheral to determine the source(s) of the interrupt. Note that multiple interrupt sources may be set in the interrupt controller—setting the IRQ or FIQ Disable flag does *not* mask interrupts from being handled by the interrupt controller, but merely prevents them from being delivered to the CPU. The firmware acknowledges any interrupts it handles. When the emulated firmware finally re-enables interrupts, a CPSR update command is sent to the target to re-enable its interrupts. If the interrupt controller still has an unacknowledged interrupt active, it will once again interrupt the target CPU. This process repeats until no interrupts are active. The acknowledgement protocol prevents any race conditions where the emulated processor may miss an interrupt. However, since these race conditions can appear natively, all ARM firmware must implement this type of protocol. Some ARM SoCs provide vectored interrupts, where the firmware can specify different handlers for each interrupt source. However, since the ARM processor itself only supports two interrupt types, these vectors are normally implemented with a small handler in ROM, which queries the interrupt controller and jumps to the correct vector. This ROM can be emulated by our system like any other firmware, allowing us to support fully-vectored interrupts with no additional work. Extracting this ROM and other per-device setup is discussed in Section 4.4.2.

<table>
<tr><th>Addr</th><th>Description</th><th>Value Specification</th></tr>
<tr><td>000</td><td>Output Control Register</td><td>Bits:
<table>
<tr><td>31-11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>Reserved</td><td>FORCEOUT</td><td>OUTEN</td><td>DBGACK</td><td>DBGRQ</td><td>nSRST</td><td>TDO</td><td>RTCK</td><td>TCK</td><td>TMS</td><td>TDI</td><td>nTRST</td></tr>
</table>
FORCEOUT – Forces JTAG output pins to the values set in this register
OUTEN – Enables JTAG output pins</td></tr>
<tr><td>004</td><td>JTAG Stream Control Register</td><td>Bits:
<table>
<tr><td>31-27</td><td>26</td><td>25</td><td>24</td><td>23-0</td></tr>
<tr><td>Reserved</td><td>Stub Interface Reset</td><td>Stub Interface Scan Enable</td><td>Stream Enable</td><td>Stream Length</td></tr>
</table>
Stub Interface Reset – Reinitializes the stub interface logic
Stub Interface Scan Enable – Causes the stub interface logic to poll the target for interrupts
Stream Enable – Streams arbitrary JTAG data (used for non-stub communication)
Stream Length – The number of bits to stream</td></tr>
<tr><td>008</td><td>JTAG Clock Divisor</td><td>Bits:
<table>
<tr><td>31</td><td>30-0</td></tr>
<tr><td>JTAG Clock Reset</td><td>Divisor</td></tr>
</table>
Divisor – The JTAG clock divisor. The JTAG clock speed is 125 MHz / (divisor – 1).</td></tr>
<tr><td>00C</td><td>Read Stall Control</td><td>Bits:
<table>
<tr><td>31</td><td>30-0</td></tr>
<tr><td>Read Stall Enable</td><td>Read Timeout</td></tr>
</table>
Read Stall Enable – Stalls reads from the Data Register until data is ready
Read Timeout – Read stall timeout, in multiples of 8 ns</td></tr>
<tr><td>x10</td><td>Read Address</td><td>Target address to read. X is the transfer size: 1 = Byte, 2 = 16 bit word, 4 = 32 bit word. Writes to this register initiate a read from the target.</td></tr>
<tr><td>x14</td><td>Write Address</td><td>Target address to write. X is the transfer size: 1 = Byte, 2 = 16 bit word, 4 = 32 bit word.</td></tr>
<tr><td>018</td><td>Data Register</td><td>Data returned from a read, or data to be written. Ignored in bulk transfer mode. Writes to this register always initiate a write to the target.</td></tr>
<tr><td>01C</td><td>IRQ Register</td><td>Bits:
<table>
<tr><td>31-8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3-0</td></tr>
<tr><td>Reserved</td><td>FIQ</td><td>IRQ</td><td>Reserved</td><td>Data Abort</td><td>Reserved</td></tr>
</table>
Reads from this register are unacknowledged exceptions received from the stub. Write a 1 back to the corresponding bit to acknowledge the exception.</td></tr>
<tr><td>024</td><td>Target CPSR</td><td>Writes to this register update the target's CPSR to the given value.</td></tr>
<tr><td>028</td><td>Bulk Data Length</td><td>Bits:
<table>
<tr><td>31-25</td><td>24</td><td>23-0</td></tr>
<tr><td>Reserved</td><td>BULKEN</td><td>Number of elements (bytes, half-words, words) to send</td></tr>
</table>
BULKEN – If set, the stub interface logic uses the bulk-optimized stub protocol, using the stub data FIFOs instead of the Data Register</td></tr>
</table>

Table 9: FPGA Register Map

►*1SXXXXXX* ►*YYYYYYYY* ◄*ZZZZZZZZ …*	*Read XX words of size S (1, 2, or 4 bytes) from address YY. XX data elements ZZ are returned.*
►2S00XXXX ►YYYYYYYY	Write a single word XX of size S (1 or 2 bytes) to address YY.
►3SXXXXXX ►YYYYYYYY ►ZZZZZZZZ …	Write XX words of size S (1, 2, or 4 bytes) to address YY. XX data elements ZZ are sent.
►50XXXXXX	Set the CPSR register to XX. Primarily used to set and clear interrupt flags.
… ◄C347A5XX …	An interrupt of type XX has occurred. This word can be sent at any time, including before a read response. In the unlikely case that a word C347A5XX is the result of a read operation, C347A500 is sent as an escape sequence.

Table 10: Stub Protocol, as 32-bit hex words

4.3.3. The Software

We modified QEMU [**93**] to pass all MMIO to our hardware. We accomplished this by creating a new "surrogate" peripheral in QEMU, which owns the entire MMIO address space of the target and forwards MMIO operations to the hardware. We also created a new QEMU "system," which selects the proper CPU, creates the necessary address spaces, initializes the surrogate peripheral, and loads the firmware to emulate.

Initially we ran our system under Windows to take advantage of the existing drivers for the PCIe card. However, the drivers were optimized for streams of data, where latency is less of a concern that throughput. For example, transfers to the card would always use DMA, regardless of the transfer size.

We ultimately re-implemented a simplified version of the driver on Linux (which was based on an open-source driver for Pico Computing's other FPGA products). To avoid syscall overhead on every MMIO operation, we allowed applications to mmap the hardware's register space, although in practice this did not significantly improve performance.

Finally, we extended the driver's interrupt handler to deliver a signal any process that requests it whenever a non-DMA interrupt is received. A signal handler in QEMU delivers this interrupt to the virtual CPU. This provides a low-latency path for interrupts.

4.4. Evaluation

We evaluate our system against two metrics: its performance and the ease of configuring it to work with a new target device.

4.4.1. Performance

One of the key motivations for SURROGATES was to overcome the performance limitations of Avatar. While we had independently built a system very similar to Avatar, we were unable to use it against several devices of interest, such as medical devices, because proper operation of many devices relies on timing constraints that it could not meet (e.g., watchdogs on co-processors of a medical device). Therefore, we evaluate the several performance aspects of our system and compare it with prior approaches.

To test raw MMIO performance, we measure the time needed to make 1,000,000 read or write requests to the SRAM of our S3C2440, connected to our hardware with via JTAG with a 4 MHz clock. We find that our raw MMIO performance is four orders of magnitude faster than what the Avatar authors reported, as shown in Table 11. We also measured the time taken to write to an FPGA register 1,000,000 times. Although accessing the FPGA through an mmap interface is about 60% faster (1.4 μs vs. 2.2 μs), the overall performance gains of this additional optimization are negligible.

To evaluate whether this performance was reasonable to support near-real time emulation, we set out to boot Linux on the emulated S3C2440. To accurately measure the amount of time to boot, we replaced the init binary with one that simply contains a special illegal instruction. This instruction shuts down QEMU and reports performance statistics. We found that the kernel boots in about 27 seconds. 25 seconds were spent performing I/O. However, during boot the kernel initializes all of the peripherals, so its I/O characteristics are different from typical usage of a booted system. During this time, approximately 126,000 reads and 87,000 writes were performed. To measure interactivity, we replaced the init binary with the busybox [**94**] version of /bin/sh. While file system accesses were noticeably slower than on the real hardware, the shell maintained subjectively good responsiveness.

4.4.2. Portability

This work was also motivated by our desire to build a dynamic analysis tool that does not require a great deal of work to apply to a new target. Therefore, we evaluate the ease of supporting new devices and discuss some of the new challenges encountered when supporting entire systems. We look at two devices as case studies: a FriendlyARM Mini2440 development board with a Samsung S3C2440 SoC, and a wireless medical device with an iMX21 SoC.

When applying our system to a new target, the first task is to identify the target's JTAG port. These are often connected to test pads on the target's PCB, but sometimes they are brought out to dedicated connectors. As a development board, the FriendlyARM had a well-identified JTAG port. The wireless medical device, however, just had dozens of unmarked test points. We had previously identified the JTAG test points through manual analysis; however, today there are tools like the JTAGulator [**95**] that perform a brute-force search over all test points to find the JTAG signals.

Once JTAG connectivity is established, firmware of the device is downloaded. In some cases, the SoC itself has a small amount of firmware in ROM that is essential to proper operation of the SoC. For example, the ROM in the iMX21 performs interrupt vectoring, so if the firmware chooses to use vectored interrupts, the ROM must be emulated as well.

A location for the stub must be identified. Different SoCs have varying requirements for locating interrupt and exception handlers. For example, on the S3C2440, exception handlers must be located at 0x00000000, while on the iMX21, we can place exception handlers anywhere in memory because the ROM at 0x00000000 uses an exception vector table stored in dedicated RAM as a level of indirection. On the S3C2440, we place our stub in the "NAND SteppingStone" SRAM at 0x00000000. On the iMX21, we place our stub in the dedicated exception handler SRAM. Depending on the SoC, it may also be possible to lock the stub into the L1 cache, allowing one to virtually overlay address spaces that are normally not usable (such as ROMs at 0). MMUs, if available, may also be used to place the stub at arbitrary locations, but this is left for future work.

Next, the layout of the target's address space must be specified in QEMU. Usually this is as simple as defining the address regions of RAM, Flash, and peripherals. For the iMX21, an additional address space entry is created for the ROM.

There are usually a few exceptions that must be carved out of the peripheral address space. These are for registers that, when updated, cause the target to lose sync with the host. For example, on the S3C2440, there are registers that control the core clock speed. When the clock speed is adjusted, the CPU is halted until the PLLs re-lock. JTAG communication fails until the CPU resumes execution. We can use dynamic analyses techniques to easily determine these exceptions. If we log all MMIO as

the system boots, the last MMIO operation before the system halts is usually responsible for the failure. The SoC datasheet can be consulted for the effect of the corresponding register so that an intelligent exception can be made.

	MMIO Operations Per Second
Avatar	5 (over serial debug port at 38400 bps)
Our system w/ syscalls	17172 writes / 15761 reads (over 4 MHz JTAG)
Our system w/ mmap	17174 writes / 15772 reads (over 4 MHz JTAG)

Table 11: Raw MMIO Performance

Finally, different SoCs have drastically varying DMA controllers, some of which must be emulated for proper emulation of the device. For example, the S3C2440 has a general-purpose DMA controller as well as a dedicate LCD DMA controller. Neither are *required* to be emulated to boot Linux. For the iMX21, we emulated the LCD DMA controller registers in QEMU with only eight additional lines of C. This emulated DMA controller simply copies the specified video memory from the emulator to the same location on the target, and then passes the DMA request on to the real DMA controller to transfer the data to the LCD.

As an alternative to emulating different DMA controllers, we can treat the emulator's memory as another level of cache. DMA controllers typically cannot access the L1 or L2 caches, so any data involved in a transfer must reside in main memory. We can treat intentional cache invalidations as an indication that the memory was or will be used in a DMA transfer and flush the affected memory to or from the target. (Note that the stub always runs with the target's data caches off, so flushes from the emulator to the target will go directly to main memory). Unfortunately, this approach only works with firmware that turns the data caches on, which was not the case with our wireless medical device.

Overall, we find it straightforward to apply our system to different devices, requiring far less work than building an emulator for all of the target's hardware. There is some manual configuration involved, but this is true of most dynamic analysis tools.

Chapter 5: Conclusions

Mark Weiser's vision of ubiquitous computing [**96**] has not only arrived, it has been surpassed, with computers now being far more embedded and ubiquitous than originally envisioned. Unfortunately, this rapid computerization of everyday things has relegated security mostly to an afterthought.

To understand the current state and challenges of embedded systems security, we thoroughly analyzed a modern automobile, a complex cyber-physical system incorporating dozens of embedded systems of varying complexity. We find that automotive systems, while tolerant of accidental failures, do not withstand malicious, *intentional* failures. In particular, we find that with access to a car's internal networks, it is relatively straight-forward for an attacker to take complete control of critical vehicle systems, including lights, locks, the engine, and brakes.

Compounding these problems is the multitude of vulnerable entry points into automotive networks. We systematically evaluate the attack surface of modern vehicles by first analyzing potential entry points across different classes of access (indirect physical, short-range wireless, and long-range wireless), and then demonstrating real vulnerabilities for each of these classes. In particular, we show complete vehicle compromises through a malicious media file, an infected dealer maintenance tool, Bluetooth, and cellular connections.

We find that these embedded systems are plagued with "old" vulnerabilities, such as buffer overflows. While there are tools to help discover and mitigate these types of vulnerabilities, there are a number of challenges in applying these tools to embedded systems. In particular, dynamic analysis tools rely on instrumenting program execution, which is difficult on embedded systems.

We have built and evaluated a system, called SURROGATES, that enables dynamic analysis of embedded systems at an unprecedented scale. Our approach is similar to Avatar; we run the system under emulation in QEMU and redirect I/O to the target hardware to guide execution and provide the firmware with a faithful reproduction of its environment. However, by using a custom FPGA bridge between the host and target, we enable near-real time emulation of the target system, allowing us to analyze systems of far greater complexity.

While our system enables dynamic analysis of embedded systems at an unprecedented scale, it does not necessarily scale any further. Systems like SAGE depend on the ability to massively parallelize state space searches. This is easy with well-defined OS APIs, but our approach depends on an individual physical system to guide execution. However, it may be possible to learn models of the hardware based on execution traces collected with our system. This would enable dynamic analysis systems to run largely independent of physical hardware, allowing it to scale up massively. The models do not necessarily need to be 100% accurate; as long as they reasonably constrain the state space search, it is feasible to explore several potentially vulnerable code paths. When a potentially vulnerable code path is found, it can be verified against the actual hardware using our system.

Given the vulnerabilities uncovered in Chapter 3, we believe a system like SAGE would be quite effective at identifying many low-hanging vulnerabilities in embedded systems. By enabling near-real-time instrumentation of embedded systems, our work provides a critical piece in enabling sophisticated dynamic analysis tools to work against embedded systems.

Bibliography

[1] David E. Sanger, "Obama Order Sped Up Wave of Cyberattacks Against Iran," *The New York Times*, June 2012.

[2] John Markoff, "Stung by Security Flaws, Microsoft Makes Software Safety a Top Goal," *The New York Times*, January 2002.

[3] Craig Mundie, Pierre de Vries, Peter Haynes, and Matt Corwine. (2002) [Online]. http://download.microsoft.com/download/a/f/2/af22fd56-7f19-47aa-8167-4b1d73cd3c57/twc_mundie.doc

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

[5] Robert Charette, "This Car Runs on Code," *IEEE Spectrum*, February 2009. [Online]. http://www.spectrum.ieee.org/feb09/7649

[6] 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.

[7] Automatic Labs, Inc., Automatic, https://www.automatic.com/.

[8] Navigation.com. Mining the Data Port II: A Look at Usage-Based Insurance Programs. [Online]. http://allthingsnav.navigation.com/article/mining-data-port-ii-look-usage-based-insurance-programs

[9] Steve Mollman. (2009, October) CNN.com. [Online]. http://www.cnn.com/2009/TECH/10/08/apps.realworld/

[10] Antuan Goodwin. (2009, October) CNET. [Online]. http://www.cnet.com/news/ford-unveils-open-source-sync-developer-platform/

[11] CAMP Vehicle Safety Communications 2 Consortium, Cooperative Intersection Collision Avoidance System, October 2008, Online: http://www.nhtsa.dot.gov/staticfiles/DOT/NHTSA/NRD/Multimedia/PDFs/Crash%20Avoidance/2008/811048.pdf.

[12] CAMP Vehicle Safety Communications 2 Consortium, Vehicle Safety Communications --- Applications First, September 2008, Online: http://www.intellidriveusa.org/documents/09042008-vsc-a-report.pdf.

[13] CAMP Vehicle Safety Communications Consortium, Vehicle Safety Communications Project Task 3 Final Report, March 2005, Online: http://www.intellidriveusa.org/documents/vehicle-safety.pdf.

[14] Virginia Tech Transportation Institute, Intersection Collision Avoidance --- Violation, April 2007, Online: http://www.intellidriveusa.org/documents/final-report-04-2007.pdf.

[15] Martin Melosi, The Automobile and the Environment in American History, 2004.

[16] Andru Edwards. (2009, March) Gear Live. [Online]. http://www.gearlive.com/news/article/q109-exclusive-twitter-integration-coming-to-onstar/

[17] Paul Eisenstein, "GM Hy-Wire Drive-By-Wire Hybrid Fuel Cell," *Popular Mechanics*, August 2002.

[18] *ISO 11898-1:2003. Road vehicles -- Controller area network (CAN).*: ISO, Geneva, Switzerland.

[19] FlexRay Consortium, FlexRay Communications System Protocol Specification, December 2005, Online: http://www.flexray.com/index.php?pid=47.

[20] Autosar, AUTomotive Open System ARchitecture, http://www.autosar.org/.

[21] Ulf E. Larson and Dennis K. Nilsson, "Securing vehicles against cyber attacks," in *CSIIRW*, May 2008, pp. 30:1--30:3.

[22] Peter R. Thorn and C. Arthur MacCarley, "A Spy Under the Hood: Controlling Risk and Automotive EDR," *Risk Management*, February 2008.

[23] Marko Wolf, A. Weimerskirch, and Christof Paar, "Security in automotive bus systems," in *ESCAR*, November 2004.

[24] Marko Wolf, André Weimerskirch, and Thomas Wollinger, "State of the Art: Embedding Security in Vehicles," *EURASIP Journal on Embedded Systems* , 2007.

[25] Yilin Zhao, "Telematics: safe and fun driving," *Intelligent Systems, IEEE*, vol. 17, no. 1, pp. 10-14, Jan./Feb. 2002.

[26] Tobias Hoppe, Stefan Kiltz, and Jana Dittmann, "Security Threats to Automotive CAN Networks -- Practical Examples and Selected Short-Term Countermeasures," in *SAFECOMP 2008*, vol. 5219, September 2008, pp. 235-248.

[27] Ishtiaq Roufa et al., "Security and Privacy Vulnerabilities of In-Car Wireless Networks: A Tire Pressure Monitoring System Case Study," Rutgers, Technical Report 2010.

[28] Aurelien Francillon, Boris Danev, and Srdjan Capkun, "Relay Attacks on Passive Keyless Entry and Start Systems in Modern Cars," in *NDSS 2011*, Feb 2011.

[29] Stephen Bono et al., "Security Analysis of a Cryptographically-Enabled RFID Device," in *USENIX Security 2005*, July 2005, pp. 1-16.

[30] Sebastiaan Indesteege, Nathan Keller, Orr Dunkelman, Eli Biham, and Bart Preneel, "A Practical Attack on KeeLoq," in *Eurocrypt '08*, vol. 4965, Spril 2008, pp. 1-18.

[31] Thomas Eisenbarth, Timo Kasper, Amir Moradi, Christof Paar, and Mahmoud and Manzuri Shalmani, Mohammad Salmasizadeh, "On the Power of Power Analysis in the Real World: A complete break of the KeeLoq code hopping scheme.," in *Crypto*, vol. 5157, August 2008, pp. 203-220.

[32] Maxim Raya and Jean-Pierre Hubaux, "Securing Vehicular Ad Hoc Networks," *Journal of Computer Security*, vol. 15, no. 1, pp. 39-68, 2007.

[33] Frank Kargl et al., "Secure vehicular communication systems: implementation, performance, and research challenges," *IEEE Communications Magazine*, vol. 46, no. 11, pp. 110-118, 2008.

[34] Mark Mansur, TunerPro - Professional Automobile Tuning Software, http://www.tunerpro.net/.

[35] Link Engine Mangement Systems, PCLink - Link ECU Tuning Software, http://www.linkecu.com/pclink/PCLink.

[36] Karl Koscher et al., "Experimental Security Analysis of a Modern Automobile," in *IEEE Symposium on Security and Privacy*, Oakland, 2010.

[37] Stephen Checkoway et al., "Comprehensive Experimental Analyses of Automotive Attack Surfaces," in *USENIX Security Symposium*, San Francisco, 2011.

[38] Charlie Miller and Chris Valasek, "Adventures in Automotive Networks and Control Units," 2013. [Online]. http://illmatics.com/car_hacking.pdf

[39] Charlie Miller and Chris Valasek, "A Survey of Remote Automotive Attack Surfaces," 2014. [Online]. http://www.ioactive.com/pdfs/Remote_Automotive_Attack_Surfaces.pdf

[40] CanBusHack. [Online]. http://canbushack.com/

[41] Vehicle Reverse Engineering Wiki. [Online]. http://vehicle-reverse-engineering.wikia.com/

[42] Eric Evenchick. (2013) Hack A Day. [Online]. http://hackaday.com/2013/10/21/can-hacking-introductions/

[43] The OpenXC Platform. [Online]. http://openxcplatform.com/

[44] Andrew Righter and Travis Goodspeed, GoodThopter, http://goodfet.sourceforge.net/hardware/goodthopter12/.

[45] Derek Kuschel, CANBus Triple, http://canb.us/.

[46] DARPA, Grand Challenge, http://www.darpa.mil/grandchallenge/index.asp.

[47] John Markoff, "Google Cars Drive Themselves, in Traffic," *The New York Times*, October 2010.

[48] Ross Anderson and Kuhn Markus, "Tamper Resistance--A Cautionary Note," in *The Second USENIX Workshop on Electronic Commerce Proceedings*, Oakland, 1996, pp. 1-11.

[49] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady, "Security in Embedded Systems: Design Challenges," *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 3, pp. 461-491, 2004.

[50] Dan Boneh, Richard A. Demillo, and Richard J. Lipton, "On the Importance of Eliminating Errors in Cryptographic Computations," *Journal of Cryptology*, vol. 14, pp. 101-119, 2001.

[51] Acoustic Cryptanalysis: On nosy people and noisy machines. [Online]. http://tau.ac.il/~tromer/acoustic/

[52] J. Ferrigno and M. Hlavac, "When AES blinks: introducing optical side channel," *IET Information Security*, vol. 2, no. 3, pp. 94-98, 2008.

[53] Paul C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *16th Annual International Cryptology Conference on Advances in Cryptology*, Santa Barbara, 1996, pp. 104-113.

[54] D. Boneh and D. Brumley, "Remote timing attacks are practical," in *12th Usenix Security Symposium*, 2003.

[55] The OpenSSL Project, OpenSSL, https://www.openssl.org/.

[56] Paul Kocher, Joshua Jaffe, and Benjamin Jun, "Differential Power Analysis," San Francisco, 1998.

[57] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi, "Introduction to differential power analysis," *Journal of Cryptographic Engineering*, vol. 1, no. 1, pp. 5-26, 2011.

[58] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach., "Analysis of an electronic voting system ," in *IEEE Symposium on Security and Privacy*, Oakland, 2004.

[59] Stephen Checkoway et al., "Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage," in *EVT/WOTE*, Montreal, 2009.

[60] T. Scott Saponas, Jonathan Lester, Carl Hartung, Sameer Agarwal, and Tadayoshi Kohno, "Devices That Tell On You: Privacy Trends in Consumer Ubiquitous Computing," in *16th USENIX Security Symposium*, Boston, 2007.

[61] D. Halperin et al., "Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses," in *IEEE Symposium on Security and Privacy*, Oakland, 2008.

[62] J. R. Smith, A. P. Sample, P. S. Powledge, S. Roy, and A. Mamishev, "A wirelessly-powered platform for sensing and computation," in *In 8th International Conference on Ubiquitous Computing*, Orange County, 2006.

[63] Hristo Bojinov, Elie Bursztein, and Dan Boneh, "XCS: Cross Channel Scripting and its Impact on Web Applications," in *Proceedings of the 16th ACM Conference on Computing and Communications Security (CCS '09)*, New York, 2009.

[64] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarott, "A Large-Scale Analysis of the Security of Embedded Systems," in *The Proceedings of the 23rd USENIX Security Symposium*, San Diego, 2014.

[65] Michael Lynn, "Cisco IOS Shellcode," in *Blackhat USA*, Las Vegas, 2005.

[66] Ellen Messmer, "Furor over Cisco IOS router exploit erupts at Black Hat," *Network World*, July 2005.

[67] Cory Doctorow, "Michael Lynn's controversial Cisco security presentation," *Boing Boing*, July 2005. [Online]. http://boingboing.net/2005/07/29/michael-lynns-contro.html

[68] Barnaby Jack, "Jackpotting Automated Teller Machines Redux," in *Black Hat USA*, Las Vegas, 2010.

[69] WiiBrew.org. [Online]. http://wiibrew.org/wiki/Signing_bug

[70] Bruce Schneier. (2011, January) Sony PS3 Security Broken. [Online]. http://www.schneier.com/blog/archives/2011/01/sony_ps3_securi.html

[71] R. Graham and D. Maynor, "SCADA Security and Terrorism: We're not crying wolf," in *Blackhat Federal*, DC, 2006.

[72] Barton P. Miller, Lars Fredriksen, and Bryan So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32-44, 1990.

[73] Michael Sutton and Adam Greene, "The Art of File Format Fuzzing," in *Black Hat USA*, Las Vegas, 2005.

[74] Dave Aitel, "An Introduction to SPIKE, the Fuzzer Creation Kit," in *Black Hat USA*, Las Vegas, 2002.

[75] Deja vu Security, Peach Fuzzing Platform, http://peachfuzzer.com/.

[76] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler, "EXE: Automatically Generating Inputs of Death," in *ACM Conference on Computer and Communications Security*, Alexandria, 2006.

[77] Patrice Godefroid, Nils Klarlund, and Koushik Sen, "DART: Directed Automated Random Testing," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, 2005.

[78] Patrice Godefroid, Michael Y. Levin, and David Molnar, "Automated Whitebox Fuzz Testing," in *The 15th Annual Network & Distributed System Security Conference*, San Diego, 2008.

[79] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley, "All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)," in *IEE Symposium on Security and Privacy*, Oakland, 2010.

[80] The Valgrind Developers, Valgrind, http://www.valgrind.org/.

[81] Drew Davidson, Benjamin Moench, Somesh Jha, and Thomas Ristenpart, "FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution," in *USENIX Security Symposium*, 2013.

[82] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti, "Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares," in *Network and Distributed System Security Symposium*, 2014.

[83] Anthony Bellissimo, John Burgess, and Kevin Fu, "Secure Software Updates: Disappointments and New Challenges," in *HotSec 2006*, Juky 2006, pp. 37-43.

[84] BBC. (2010, May) BBC News. [Online]. http://www.bbc.co.uk/news/10119492

[85] Rebecca Boyle, "Proof-of-Concept CarShark Software Hacks Car Computers, Shutting Down Brakes, Engines, and More," *Popular Science*, May 2010.

[86] John Leyden. (2010, May) The Register. [Online]. http://www.theregister.co.uk/2010/05/14/car_security_risks/

[87] K. Koscher et al., "Experimental Security Analysis of a Modern Automobile," in *IEEE Symposium on Security and Privacy*, May 2010.

[88] NPR. (2005, April) NPR News. [Online]. http://www.npr.org/templates/story/story.php?storyId=4599106

[89] Nicolas Falliere, Liam O Murchu, and Eric Chien, W32.Stuxnet Dossier Version 1.3, November 2010, Online: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

[90] Jaikumar Vijayan. (2010, August) Computerworld. [Online]. http://www.computerworld.com/article/2515079/security0/update--android-gaming-app-hides-trojan--security-vendors-warn.html

[91] Dominic Spill and Andrea Bittau, "BlueSniff: Eve meets Alice and Bluetooth," in *USENIX Workshop on Offensive Technologies*, 2007, pp. 1-10.

[92] OpenOCD. [Online]. http://openocd.sourceforge.net/

[93] F. Bellard, et. al. QEMU. [Online]. http://www.qemu.org/

[94] BusyBox, http://www.busybox.net/.

[95] Joe Grand. (2013) JTAGulator. [Online]. http://www.grandideastudio.com/portfolio/jtagulator/

[96] Mark Weiser, "The Computer for the 21st Century," *Scientific American*, vol. 265, no. 3, pp. 94-104, September 1991.