Secure Automotive Software

The Next Steps

Lee Pike, Jamey Sharp, Mark Tullsen, Patrick C. Hickey, and James Bielman

Previous research revealed pervasive software vulnerabilities in modern automobiles. This article presents a rejoinder to that research, discussing four general approaches to secure automotive software systems: compile-time assurance, runtime protection, automated testing, and architecture security.

Author's version. Published version appears in IEEE Software (Volume: 34, Issue: 3, May-Jun. 2017)

In 2010 and 2011, a research team from the University of California, San Diego and University of Washington (hereafter called UCSD–UW) showed that modern cars can be compromised remotely through exploitation of software-based vulnerabilities.[1,2] Modern cars have a lot of software to hack. Estimates suggest that a luxury car contains tens of millions of LOC executing on 50 to 70 electronic control units (ECUs). An economy car might contain more than 25 ECUs, and the number of ECUs continues to grow.

Motivated by these sort of attacks, other security concerns,[3] and more general automotive software engineering challenges,[4] organizations have provided some basic guidance on improving automotive cybersecurity. The SAE J3061 standard, released in early 2016, offers high-level guidance, focusing primarily on processes.[5] Also in 2016, the US National Highway Transportation Safety Administration released a draft set of best practices.[6]

Here, we provide more specific recommendations on securing software systems. We focus on software using artifact-based approaches rather than processes. Our recommendations are based on concrete evidence about what works and what doesn't. Regarding what doesn't work, we draw lessons from successful car-hacking attempts, such as the UCSD–UW research.

Regarding what works, we draw lessons primarily from our experiences in DARPA's High-Assurance Cyber Military Systems (HACMS) program.[7] In that program, we and many collaborators built tools to construct cyberattack-resistant cyber-physical systems. These systems included ground robots, automobiles, manned aircraft, and unmanned aerial vehicles (UAVs). Our team focused on air vehicles. The independent US-government-sponsored red team that analyzed the UAV found it to be free from typical software-based vulnerabilities. One government official deemed it "the most secure UAV on the planet."[8] Later, we transitioned the tools to Boeing for them to secure the software in the optionally unpiloted Little Bird helicopter.

The USCD–UW Attacks

The key enabler of the UCSD–UW attacks was the ease of accessing at least one of a car's data buses. These buses let the ECUs coordinate with each other—for example, so that the braking system can interact with the engine controllers to provide better control. Some ECUs act as data bus bridges and can broadcast on multiple buses. In the analyzed automobiles, every ECU was transitively connected to every other ECU through a data bus or data bus bridges. This highly connected architecture was driven by complex interactions required for safety or desired for comfort. For example, the door lock system had to know when the airbags had been deployed so that it could automatically unlock all doors to make escape easier. The entertainment system had to know the vehicle's speed so that it could raise the audio volume to compensate for wind and road noise.

The UCSD–UW researchers found multiple attack vectors. These included an audio track played in the CD player, Bluetooth access to the entertainment system, and cellular access to the telematics system. The researchers also considered other vectors, such as remotely hacking a mechanic's diagnostics tool.

Each attack initially exploited some interface vulnerability. Such vulnerabilities included some combination of the brute-force guessing of short PIN numbers (for example, Bluetooth), exploiting buffer overflows in low-level networking code, shell code injections, and exploiting automated firmware updates. Once an attack accessed a data bus, it obtained further access by reprogramming other ECUs over the data bus. To do this, the attack exploited mechanisms that normally let mechanics update software without needing physical access to individual ECUs.

Automotive Challenges to Secure Software

The following impediments to improving software security are specific to the automotive industry. Alexander Pretschner and his colleagues have also covered some of these impediments in depth.[4]

Part Cost

The automotive industry is extremely sensitive to part cost, making expensive hardware-based security solutions difficult to achieve.

Size and Weight

Solutions must take into account size and weight increases. For example, replacing a Controller Area Network (a line topology) with Ethernet (a star topology with a central switch) might be infeasible owing to the extra wiring required.

Legacy Integration

The automobile industry often depends on long component lifetimes to keep costs down, so a new design might have to integrate with legacy components. Approaches requiring major architectural changes, such as changes in bus technology, might be delayed or ruled out to maintain compatibility with legacy components.

Memory Constraints

Cost pressures require that ECUs use the least expensive components that can do the job. Particularly in small microcontroller-based ECUs, memory is the costliest part. Software for such ECUs is designed to have the smallest memory footprint possible, and security approaches that use large amounts of memory might be ruled out on the basis of cost.

Timing Requirements

Many ECUs perform tasks with a fixed real-time deadline (the time between receiving and carrying out a command), which is often safety-critical. Security measures that might prevent a program from meeting timing requirements will be ruled out on the basis of safety.

Standardization

A single original equipment manufacturer (OEM) can't afford to break away from industry standards. Keeping supply chain costs low requires leveraging the suppliers that make similar parts for multiple manufacturers.

Supplier Integration

To protect intellectual property, suppliers often provide components without source code, making assessment, modification, or instrumentation by an OEM to improve security more difficult.

Dealing with the Impediments

Our recommendations try to respect these constraints. In particular, we focus on improving software quality rather than making hardware more secure. On the other hand, we're optimistic that some of today's technical constraints will become less onerous in the future. For example, as processor manufacturers retire old components, some processors will be upgraded to modern alternatives that offer superior security features "for free"—such as virtual memory or cryptographic instructions. However, the current supplier model might be the most problematic issue. As the UCSD–UW researchers noted, *While this outsourcing process might have been appropriate for purely mechanical systems, it is no longer appropriate for digital systems that have the potential for remote compromise.*[2]

Our Recommendations

The automotive industry has often ignored the low-hanging fruit for improving software quality,[9] such as using version control, unit testing, integrated testing, and code reviews. The Motor Industry Software Reliability Association's *Development Guidelines for Vehicle Based Software* already recommends these approaches,[10] so we don't discuss them here.

The following recommendations typically go beyond the automotive industry's current standard practices. Table 1 summarizes the recommendations, organized into four areas: compile-time assurance, runtime protection, automated testing, and architectural security.

Table 1. Recommendations for improving automotive-software security.

Area	Recommendations
Compile-time assurance	• Static analysis • Memory-safe programming • Formal verification
Runtime protection	• System specialization • Measurement and attestation • Cryptography • Runtime verification
Automated testing	• Fuzz testing • Property-based testing
Architectural security	• Trusted interfaces * Software isolation * Glue code generation

Compile-Time Assurance

Compile-time assurance happens before code execution. We present the recommendations in the increasing order of engineering effort.

Static analysis. Static analysis tries to discover software flaws without execution or testing, and many tools are commercially available. Some tools are *sound*; that is, they shouldn't produce false negatives. To improve scalability and reduce false positives, some tools are *unsound* and can be considered advanced bug-hunting tools.

SAE J3061 recommends using static analysis, and we do too. However, although static analysis is powerful, it can lead to a false sense of security. Furthermore, static-analysis tools can produce so many false positives that discerning legitimate vulnerabilities is difficult. Finally, static analysis is unreliable for automatically uncovering domain-specific bugs.

Memory-safe programming. Microsoft discovered that the Pareto principle applies to software quality: 80 percent of Windows and Office errors and crashes came from 20 percent of the bugs.[11] We conjecture that the principle applies more generally to software security: 80 percent of exploits come from 20 percent of the classes of vulnerabilities.

For example, all the UCSD–UW attacks on short- and long-range wireless systems depended on exploiting buffer overflows.[2] Buffer overflows are a rudimentary vulnerability known since at least 1972.[12] They're a particular example of a memory-safety violation, which is an example of undefined behavior. Coding standards and static analysis target mostly the prevention and discovery, respectively, of undefined behavior resulting from using "unsafe" programming languages such as C or C++.

We propose that the most expeditious way to improve software security is to use memory-safe languages. *Safe-C* languages are memory-safe and suitable for embedded programming. They guarantee memory safety while still allowing programmers fine-grained control of memory use and timing.

We developed the safe-C language Ivory[13] to support the HACMS program. Ivory is a secure alternative to C/C++ in which memory-safety errors are impossible; it supports a variety of verification tools. The HACMS program used Ivory to develop secure avionics with no memory-safety vulnerabilities.

Formal verification. Whereas testing provides partial assurance about the actual artifact to be fielded (because one more test vector might uncover a vulnerability), formal verification provides complete assurance about a model of the system. With testing, the designer's worry is, "Have I tested enough?" With formal verification, the worry is, "Is my model's fidelity accurate enough?"[14]

Formal verification requires a two-step approach: build a model, then verify it. Both steps are usually partly manual. Because of the effort involved, formal verification is the most cost-effective for critical, well-defined components. One example is embedded OSs.[15] Another example is specific control systems. These systems are particularly difficult to test because they combine continuous dynamics with discrete control. However, inroads are being made into formal verification of automotive control systems.[16]

Runtime Protection

Here we describe four types of runtime protection.

System specialization. In the UCSD–UW analysis, some middleware contained a full installation of an OS (for example, Linux), complete with standard root-level networking tools. The UCSD–UW researchers leveraged these tools to simplify the attacker's analysis of other parts of the system and to simplify the attacks. However, these tools didn't have to be on the system. Stephen Checkoway and his colleagues noted the following:

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

The lesson is that only the software required during deployment should be installed in deployment.

Moreover, OS researchers have been developing a unikernel approach that develops an OS and drivers as a set of specialized libraries. Only the libraries required for the applications running on the OS are linked in. One virtual machine implementing this approach is HaLVM (Haskell Lightweight Virtual Machine).[17] Tools such as HaLVM are particularly relevant for securing systems built on modern application processors, such as the infotainment system.

Measurement and attestation. Measurement and attestation (M&A) checks the value of data, including the executable, and then proves to a third party that the values are as expected.[18] M&A often assumes the existence of special hardware (such as a Trusted Platform Module) to provide a root of trust, but such hardware might not exist on small microcontrollers. So, researchers have proposed lighter-weight solutions for the automotive industry.[19]

M&A is particularly relevant to ensure that ECUs execute only the binaries the manufacturer has endorsed. Reflashing ECUs with maliciously modified binaries was a key element in some of the UCSD–UW attacks. An M&A infrastructure could make such attacks more difficult.

Cryptography. Cryptography is a common recommendation for improving security, and strong cryptography (including authentication) can make some aspects of automotive software more secure.[5] For example, firmware updates, either provided by mechanics or over the air, should be signed and encrypted.[20]

The principal security risk with cryptography is misuse in its application, not flaws in the cryptography itself.[21] The implementation details of any cryptographic solution should be reviewed by cryptography experts and tested by a team of penetration-testing experts.

Runtime verification. Runtime verification is a research field that marries formal verification and testing. The idea is to take a high-level specification of program behavior—the same specification that might be used for formal verification at compile time—and automatically instrument a program to check for conformance with the specification. Runtime verification separates software control from software monitoring and eliminates the onerous task of proving correctness at compile time.[22]

A particular challenge of runtime verification is how to instrument a program to check all control paths that are relevant to the specified property, while ensuring that the instrumentation doesn't adversely change the program's behavior (particularly the nonfunctional behavior, such as timing and memory use).

Automated Testing

Testing is the primary means in industry to provide software assurance. Two highly effective types of automated testing are *fuzz testing* and *property-based testing*.

Fuzz testing. Fuzz testing has had significant industrial adoption and is recommended by SAE J3061. It automatically "fuzzes" (randomizes) conforming inputs to discover slightly out-of-specification inputs that cause bugs. Fuzz testing is particularly effective for discovering bugs caused by insufficiently sanitized user input.

Property-based testing. Property-based testing (PBT) automatically derives test cases from the property specification and possibly a data format configuration. PBT is roughly the inverse of fuzz testing in that the tests are generally expected to be well-formed. PBT tools initially generate simple test cases and then iteratively generate more elaborate tests. Because PBT generates tests automatically, generating one million tests requires no more effort (disregarding computation time) than generating 100 tests.

One of the first PBT tools was QuickCheck. It has been applied to real-world automotive problems; for example, Volvo is using it to test conformance with the AUTOSAR (Automotive Open System Architecture) 4.0 standard.[23]

Architectural Security

SAE J3061 mentions that software partitioning and isolation are important for security. We refine those claims along three dimensions here. First, we discuss an approach for building trusted interface software. Then, we discuss isolation more generally. Finally, we discuss the problem of architectural glue code.

Trusted interfaces. Every remote attack on a vehicle requires penetrating the software in some external interface. Two important functions of interfaces are to decode and sanitize data. A data description language, such as the commonly used ASN.1 (Abstract Syntax Notation 1), provides a notation for describing messages and standards for encoding, decoding, and transmitting messages over a wired or wireless network.

Unfortunately, interface software often has critical flaws. For example, MITRE's Common Vulnerability Enumeration database (cve.mitre.org) lists 99 vulnerabilities related to software implementing ASN.1 encoders and decoders. Data description languages have complexities and idiosyncrasies that make secure encoding and decoding difficult.[24] For example, complex languages encourage ad hoc hand-written decoders, which are more error-prone than machine-generated decoders. Catch-all messages, debugging messages, and user-defined messages are also attack vectors.

Simple interfaces are secure interfaces. Of all the onboard software, interface software must be simple, well defined, and rely on no assumptions about externally defined messages.

Software isolation. A car's *head unit*, which includes the telematics and entertainment systems, is the component most likely to have an exploitable vulnerability. However, it's also the easiest to compartmentalize, for two reasons. First, it's centrally located in the vehicle, which minimizes extra wiring. Second, it uses relatively high-end CPUs with a variety of security features and support for higher-cost buses such as Ethernet.

The head unit should be isolated, and its different functions can be partitioned. Fine-grained, formally verified partitioning is possible using a microkernel such as seL4.[15] Through technologies such as unikernels, lightweight, fine-grained isolation is achievable.

Glue code generation. Glue code provides the interface between software subcomponents and between applications, the OS, and drivers. It's usually boilerplate. It can integrate a collection of reusable components into a complete system. Also, it can modify data formatting as the data passes between components and can translate new software interfaces to work with legacy components.

Glue code is conceptually simple, but it's often where errors occur because correctness relies on understanding both the requirements and assumptions of all the software components it touches. Glue code flaws have been responsible for multiple published security attacks on automobiles, including Bluetooth use, diagnostic PassThru systems (used by mechanics to diagnose and update ECUs), and even the audio system.[2]

A top-level specification describing individual components' assumptions and requirements is good engineering practice, but too often these requirements and the architectural model might diverge from the implemented system. In the HACMS program, we developed formal architectural models of the system from which we could reason about dataflow and connectivity. We then generated glue code from those models directly, tying the analysis to the implemented systems.[13,25] Doing so ensured that our requirements and models didn't drift from the implementations.

Conclusion

Automobiles' software infrastructure is rapidly expanding in functionality and complexity. This includes more driver assistance and autonomy, increasing head-unit functionality, and even vehicle-to-vehicle and vehicle-to-infrastructure communication.[26] Although securing software will become more challenging, the advice laid out in this article will still provide the basis for secure software.

Even though our recommendations largely go beyond today's common practices, they're achievable with a modest increase in effort. Because automotive recalls are increasingly based on software vulnerabilities, we believe the recommendations will help achieve significant long-term cost savings.

Acknowledgments

DARPA contract HR0011-14-C-0113 partly supported this research. All opinions expressed herein are our own. This article is based on a Galois technical report.[27]

References

1. K. Koscher et al., “Experimental Security Analysis of a Modern Automobile,” *Proc. 2010 IEEE Symp. Security and Privacy* (SP 10), 2010, pp. 447–462.
2. S. Checkoway et al., “Comprehensive Experimental Analyses of Automotive Attack Surfaces,” *Proc. 20th USENIX Conf. Security* (SEC 11), 2011, p. 6.
3. F. Sagstetter et al., “Security Challenges in Automotive Hardware/Software Architecture Design,” *Proc. 2013 Conf. Design, Automation and Test in Europe* (DATE 13), 2013, pp. 458–463.
4. A. Pretschner et al., “Software Engineering for Automotive Systems: A Roadmap,” *Proc. 2007 Future of Software Eng.* (FOSE 07), 2007, pp. 55–71.
5. *J3061: Cybersecurity Guidebook for Cyber-Physical Vehicle Systems*, SAE Int’l, 2016.
6. *Cybersecurity Best Practices for Modern Vehicles*, report DOT HS 812 333, US Nat’l Highway Transportation Safety Administration, 2016.
7. L. Pike, “Hints for High-Assurance Cyber-Physical System Design,” *Proc. IEEE Cybersecurity Development* (SecDev 16), 2016; doi:10.1109/SecDev.2016.016.
8. K. Fisher, “Using Formal Methods to Enable More Secure Vehicles: DARPA’s HACMS Program,” *Proc. ACM SIGPLAN Int’l Conf. Functional Programming* (ICFP 14), 2014, p. 1.
9. P. Koopman, “A Case Study of Toyota Unintended Acceleration and Software Safety,” 2014; users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf.
10. *Development Guidelines for Vehicle Based Software*, Motor Industry Software Reliability Assoc., 1994.
11. P. Rooney, “Microsoft’s CEO: 80–20 Rule Applies to Bugs, Not Just Features,” CRN, 3 Oct. 2002; www.crn.com/news/security/18821726/microsofts-ceo-80-20-rule-applies-to-bugs-not-just-features.htm.
12. J.P. Anderson, *Computer Security Technology Planning Study*, ESD-TR-73-51, vol. 2, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), 1972.
13. P.C. Hickey et al., “Building Embedded Systems with Embedded DSLs,” *Proc. 19th ACM SIGPLAN Int’l Conf. Functional Programming* (ICFP 14), 2014, pp. 3–9.
14. J. Hurd, “Formally Verified Endgame Tables,” 2013; www.gilith.com/research/talks/psu2013.pdf.
15. G. Klein et al., “seL4: Formal Verification of an OS Kernel,” *Proc. ACM SIGOPS 22nd Symp. Operating Systems Principles* (SOSP 09), 2009, pp. 207–220.
16. X. Jin et al., “Powertrain Control Verification Benchmark,” *Proc. 17th Int’l Conf. Hybrid Systems: Computation and Control*, 2014, pp. 253–262.
17. A. Wick, “The HaLVM: A Simple Platform for Simple Platforms,” presentation at 2012 Xen Summit, 2012; www.slideshare.net/xen_com_mgr/the-halvm-a-simple-platform-for-simple-platforms.
18. X. Zhang, O. Acıiçmez, and J.-P. Seifert, “Building Efficient Integrity Measurement and Attestation for Mobile Phone Platforms,” *Security and Privacy in Mobile Information and Communication Systems*, Springer, 2009, pp. 71–82.
19. F. Stumpf, *CycurHSM: An Automotive-Qualified Software Stack for Hardware Security Modules*, white paper, ESCRYPT; www.escrypt.com/fileadmin/escrypt/pdf/CycurHSM-Whitepaper.pdf.
20. J. Samuel et al., “Survivable Key Compromise in Software Update Systems,” *Proc. 17th ACM Conf. Computer and Communications Security* (CCS 10), 2010, pp. 61–72.
21. D. Lazar et al., “Why Does Cryptographic Software Fail? A Case Study and Open Problems,” *Proc. 5th Asia-Pacific Workshop Systems* (APSys 14), 2014, article 7.
22. A. Goodloe and L. Pike, *Monitoring Distributed Real-Time Systems: A Survey and Future Directions*, tech. report NASA/CR-2010-216724, NASA Langley Research Center, 2010.

23. "Checksum Property for AUTOSAR," QuviQ, 2014; www.quviq.com/checksum-property-for-autosar.
24. L. Sassaman et al., "The Halting Problems of Network Stack Insecurity," *USENIX ;login:*, vol. 36, no. 6, 2011, pp. 22–32.
25. M.W. Whalen, D.D. Cofer, and A. Gacek, "Requirements and Architectures for Secure Vehicles," *IEEE Software*, vol. 33, no. 4, 2016, pp. 22–25.
26. J. Harding et al., *Vehicle-to-Vehicle Communications: Readiness of V2V Technology for Application*, report DOT HS 812 014, US Nat'l Highway Traffic Safety Administration, 2014.
27. L. Pike et al., *Securing the Automobile: A Comprehensive Approach*, tech. report, Galois, 2015.

Lee Pike *is the director of the cyber-physical program at Galois. His research focuses on high-confidence cyber-physical systems. Pike received a PhD in computer science from Indiana University. He's a member of IEEE and ACM. Contact him at leepike@galois.com.*

Jamey Sharp *is an independent consultant and was previously a research engineer at Galois. His research interests include software reliability, systems programming, and programming languages. Sharp received a BS in computer science from Portland State University. Contact him at jamey@minilop.net.*

Mark Tullsen *is a research engineer at Galois. His research interests include functional programming, compilers, and software security. Tullsen received a PhD in computer science from Yale University. He's a member of ACM. Contact him at tullsen@galois.com.*

Patrick C. Hickey *is a software engineer at Helium, where he works on HeliumOS, an OS for Internet-of-Things devices. Previously, he worked at Galois on research programs related to secure embedded systems, including DARPA's High-Assurance Cyber Military Systems program. Hickey received a BS in electrical engineering from Rutgers University. Contact him at pat@moreproductive.org.*

James Bielman *is a security-and-privacy architect at Tozny and was previously a research engineer and project lead at Galois. His research interests include embedded systems, security, and programming languages. Contact him at jamesjb@tozny.com.*