Hints for High-Assurance Cyber-Physical System Design

Lee Pike
Galois, Inc.
leepike@galois.com

***Abstract*—With deference to Butler Lampson, I present five hints specifically for building high-assurance cyber-physical systems: (1) use Turing-*in*complete languages (2) simple interfaces are secure interfaces, (3) *program* the glue code and architecture, (4) system verification is a probabilistic game, and (5) high-assurance systems require a high-assurance culture.**

I. Introduction

Butler Lampson's 1983 "Hints for Computer System Design" describes sound engineering practices for building large and complex computing systems [1]. More than 30 years later, the advice is relevant yet sadly ignored.

With deference to Lampson, I provide hints for building high-assurance cyber-physical systems (CPS). While Lampson broadly considers computer systems, ranging from programming languages to the (beginnings of) the internet, I focus on security- and safety-critical systems; examples include aircraft, automobiles, and embedded medical devices. The domain obviates some of Lampson's advice; for example, Lampson notes that "one crash a week is usually a cheap price to pay for 20% better performance" [1], but I cannot endorse such advice for software that puts peoples' lives at stake. Moreover, high-assurance systems are often embedded systems, and the advice focuses on that domain.

The hints are drawn from high-assurance systems I have been involved in designing and building, both in government and industry. For the sake of continuity, most of the examples I draw on come from my team's work in DARPA's *High-Assurance Cyber Military Systems* (HACMS) program. The goal of the program is to demonstrate the feasibility of using formal verification to improve the software security for complex cyber-physical systems. The problem space is motivated by the insecurity of modern CPS software, such as automotive software [2], [3]. The approach of HACMS is to leverage advances in formal verification to *guarantee* a system's correctness.

My team, in collaboration with others on the program, focused on building secure autopilot software for an unpiloted air vehicle (UAV), called *SMACCMPilot* [4]. Our deadline was tight: with a team of three engineers and an eighteen month deadline for the first operational release, we built new programming languages from scratch [5], and then used them to build an autopilot. An autopilot is a bit of an understatement since we built a full system of which the core autopilot was one part: we built a board support package for new hardware, device drivers, encrypted wireless communications with a base station, arming logic, control loops, etc.

To assess our progress, not only did we provide live flight demonstrations, but an experienced "red team" was given full access to the systems, including source code, and they attempted to discover vulnerabilities. Mostly, they did not. As one government official said, our team "had likely built the most secure UAV in the world."

I give five hints:

1) use Turing-*in*complete languages (Section II)
2) simple interfaces are secure interfaces (Section III)
3) *program* the glue code and architecture (Section IV)
4) system verification is a probabilistic game (Section V)
5) high-assurance systems require a high-assurance culture (Section VI)

In addition, I talk about our failures (Section VII), for which others can hopefully provide hints in the future.

While I will give concrete examples from HACMS to illustrate my hints, this paper does not describe how to build an autopilot or the research we undertook. This paper is about how the general lessons we learned can be applied to future systems. Furthermore, this paper does not focus on formal verification despite HACMS being primarily a formal verification project. None of the hints endorse esoteric verification expertise.

As Lampson said, these are just hints, not laws, rules, principles, etc. There are exceptions and counterexamples.

Lampson decorated his hints with appropriate quotations from Shakespeare's Hamlet. Cyber-physical systems are vulnerable computers adrift in an inhospitable world, like the British children in *Lord of the Flies*; I decorate my hints with quotations from this work.

II. Constrain the Programming Languages

"The rules!" shouted Ralph, "You're breaking the rules!"

As I mentioned in the introduction, one motivation for the HACMS program was research demonstrating that vulnerabilities in modern automotive software could be exploited, giving an attacker full control over the software systems. In modern vehicles, software control means vehicle control, and the attackers were able to perform actions such as disabling breaks, tightening seat belts, and engaging the throttle, all without driver input.

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

Fig. 1. Table reproduced from Checkoway *et al.* [3].

Attacks were built to exploit vulnerabilities in every remote interface present on the vehicle, including WiFi, Bluetooth, and Cellular. A portion of the table from Checkoway *et al.* [3] is reproduced in Figure 1. The "Channel" column shows what component is exploited. The "Full Control" column shows whether the attacker has full control of the vehicle (i.e., can modify the software on any networked component in the car) after the attack; in each case, the answer is yes.

What novel and esoteric exploits were developed to penetrate the software systems? Mostly, none. The "Implemented Capability" column describes the vulnerability exploited to gain access. You see "buffer overflow" over and over. A buffer overflow occurs when an array is indexed beyond its end, allowing one to read or write into unintended memory. A buffer overflow is the vanilla ice cream of memory safety bugs; this ubiquitous class of bugs have been known since at least 1972 [6]. They are also a solved problem.

Too often, general-purpose languages used in high-assurance systems lead to unintended consequences. These languages have two classes of problems: (1) they have surprising semantics and (2) they put the burden of correctness on developers.

Regarding (1), undefined behavior, such as the result of buffer overflow, results in surprises, and in high-assurance systems, surprises are bad. With undefined behavior, the combination of program, compiler, optimization level, and hardware, does something, but it is outside of the programming language's semantics and quite likely, the programmer's intention. To a first-order approximation, all cyber-physical systems are written in C (or its cousin, C++). The claim is particularly true when all of the software is considered including the operating system, network stack, device drivers, etc. Programs written in C are the primary source of buffer overflows, and buffer overflows are just one of a litany of undefined behaviors possible in C.

But its not just undefined behavior that is problematic. C is rife with implementation-defined behavior. Implementation-defined behavior can be particularly surprising in systems that are implemented on microcontrollers that may not have 32 or 64 bit architectures.

I would argue that even *defined* C is problematic. For example, consider the following code snippet, inspired by a genuine bug my team discovered when porting an existing open-source autopilot written in C from an 8-bit architecture to a 32-bit architecture:

```
uint8_t a = 10;
uint8_t b = 250;
printf( ``Answer: %i, %i'', a-b > 0
      , (uint8_t)(a-b) > 0
      );
```

The result is "`0, 1`". Why? Because arithmetic is *defined* in C only for values at least as wide as an `int`. So any value with a type that is smaller (e.g., a byte) is promoted, and because an `int` is signed, $10 - 250 = -240$ is less than 0. But in the second expression, we cast -240 back to an `uint8_t`, which is greater than 0. Even defined C is not a simple language.

Regarding (2), general-purpose languages unduly burden the developer to ensure correctness. General-purpose languages are Turing-complete languages that allow arbitrary computation. Specifically, a Turing-complete language allows unbounded memory allocation and nonterminating computations. Rice's Theorem guarantees there is no decision procedure for a (non-trivial) property of Turing-complete languages [7]. This means that the verification of Turing-complete languages requires heuristics or human insight. Famously, there is no decision procedure for termination.

But for less powerful languages, decision procedures for interesting properties are possible. For example, synchronous dataflow languages, like Lustre [8], guarantee bounded memory and bounded computation time. For real-time control systems, Turing completeness is not often necessary. Rather than being disappointed by Rice's Theorem, we should celebrate its converse: that for sufficiently weak languages, automated

verification is feasible. Safety-critical software best practices essentially mandate the use of a decidable fragment inside of general-purpose languages; see Gerald Holzman's "Power of 10", used by the Jet Propulsion Laboratory to develop space software, for example [9].

The lack of power that comes with Turing-incompleteness can be mitigated by a Turing-complete macro language that allows arbitrary computation at compile-time, while still guaranteeing that the generated program satisfies any required properties. For example, the Ivory language developed and used on HACMS uses a type-safe macro system to simulate general-purpose programming at *compile-time* while guaranteeing the safety of the generated code [5].

Hint: use Turing-incomplete languages with simple, unsurprising semantics.

III. Secure the Interfaces

The world, that understandable and lawful world, was slipping away.

If someone wants to rob your house, they must first break into it. To break in, one searches for the easiest entry point to circumvent; a lock on the door is useless if there is an open window. Cyber-physical systems are no different. As seen in Figure 1, there are a lot of external interfaces on a modern automobile. Autonomous and connected cars have even more. Each interface contains 10s or 100s of thousands of lines of code, operating all the way from the physical layer to the application layer.

Many security vulnerabilities result from poor interface design, or rather, lack of design. Lampson's original advice on building simple and predictable interfaces [1] is even more critical in the CPS domain. Still, Lampson's advice is often ignored.

A few anti-patterns in embedded interfaces are particularly troublesome. One is the "catch-all" message, designed for a payload not explicitly supported by the protocol. For example, it might be used for one-off debugging. Another anti-pattern is allowing user-defined message types or user-defined payload sizes. For example, one unpiloted air vehicle vulnerability resulted from the ability to upload multiple waypoints used by the vehicle to develop a flight plan. The waypoints are stored in a buffer on the vehicle, and it was assumed that no flight plan would contain more than a small fixed number of waypoints. While a reasonable assumption, there was no check on the constraint, allowing the buffer storing waypoints to overflow.

Moreover, do not conflate maintenance with operation. The remote exploits described in Section II essentially depend on what is a maintenance mode in which the embedded processors can be reflashed over the CAN bus. Similarly, the open-source ArduPilot autopilot[1] provided the ability to reflash the autopilot via the telemetry as a convenience to users.

Expressive protocol languages are more likely to have implementation flaws. Sassaman *et al.* forcefully makes the point from a language-theoretic perspective [10]. I described the security implications of Rice's Theorem in Section II, and these implications hold for interface and protocol design, too. In particular, for interfaces that can be described using context-free grammars, serializers (or encoders) and deserializers (or decoders) can be easily generated from a specification.

Hint: simple interfaces are secure interfaces.

IV. Automate the Tedium

"Ralph . . . would treat the day's decisions as though he were playing chess. The only trouble was that he would never be a very good chess player."

For years I calculated my own U.S. federal taxes by manually filling out government forms. And for years, the revenue service would amend mistakes I had made, often in my favor! The mistakes were almost always simple calculation errors. If a few dozen lines of addition and subtraction were too much to get right, I have no hope of managing the tedium of building secure CPS.

Examples of computational tedium include memory management, parsing/serialization, concurrency, inter-process communication, and system configuration. More generally, critical *computational tedious* codes have the following characteristics:

- they are pervasive throughout the system,
- they have regular usage patterns, perhaps with a small number of exceptions,
- and getting them wrong causes systematic failure.

The pitfalls of manual approaches to automate memory management and parsing/serialization are described in Sections II and III, respectively.

Concurrency bugs—such as deadlocks, livelocks, and priority inversion—are notorious as well; consider the Mars Pathfinder concurrency bug [11]. The first solution to concurrency is not to use it whenever possible; for example, by using cooperative scheduling. But sometime preemptive scheduling is necessary, although it is not an excuse for ad-hoc concurrency.

On SMACCMPilot, we built a domain-specific concurrency language called *Tower*. Tower follows a Hoare monitor model [12], in which *methods* that share state are encapsulated within a *monitor*. In a monitor, only a single thread may be active (by calling a single method) at a time. A method may invoke one more more additional methods, perhaps in different monitors, via channels.

The semantics guarantee that deadlocks are not possible, albeit it disallows some communication patterns. Monitors cannot be nested, so with a simple priority inheritance or priority ceiling protocol, priority inversion is not possible, either.

From a single Tower specification, "glue code" for FreeRTOS,[2] POSIX, and seL4 [13] is generated. The software marshaling between interfaces is a kind of "glue code" that does not implement behavior itself but just connects components.

[1] ardupilot.org

[2] http://www.freertos.org/

While glue code is rote, it is security critical. As noted in the car hacking work by Checkoway *et al.*, "virtually all vulnerabilities emerged at the interface boundaries between code written by distinct organizations" [3].

Tower is also a Turing-*in*complete language with a Turing-complete macro language. Arbitrary computation can be done at compile-time, but once the architecture is generated, it is fixed. Tower shares a type system with Ivory, mentioned in Section II.

An integrated configuration language allows powerful reconfigurations. For example, the current SMACCMPilot implementation is a two-board system. One board is the *mission board* that performs ground communication, crypto, and hosts high-level mission software; the other board is the flight management unit (FMU) that performs inertial navigation and motor control. In testing, we sometimes wish to execute the system as a simpler single-board system on the FMU. Building a single-board architecture that moves communication components to the FMU and elides the databus between the FMU and mission computer is as easy as running the Tower program with new arguments.

The ability to refactor an architecture like a program changes one's perspective of a system architecture as being a fixed monolithic application to a set of libraries that can be composed in various ways.

The guiding theme is that as computer scientists, we know how to build abstractions and computations with programming languages. High-level, safe languages specialized for managing concurrency, communication, and configuration are just as important as languages for building individual behavioral components.

Hint: program the glue code and architecture.

V. The Verifier's Dilemma

"We've got to have rules and obey them. After all, we're not savages."

A Google employee remarked once that new tools to automatically discover new bugs are not of interest because they—just like other large software corporations—already have more open bugs than they have time to fix. These bugs exist in both open-source software that the companies depend on (e.g., Linux, OpenSSL, etc.) and proprietary software alike. Rather than discovering new bugs, their need is to triage existing bugs quickly, to discover which vulnerabilities are exploitable from external interfaces; those are the critical bugs.

So if the goal is to reduce the probability of exploitable bugs, the first task is to understand which components are security critical by carrying out an architectural analysis. Assuming one has a handle on which components are critical, a dilemma presents itself: should resources be dedicated to absolutely assuring the correctness of the critical components, or should they be spread across the system? The choice is the verifier's dilemma. Uncritical components can become critical if the critical ones are not correct. Like in poker, the "pot odds" have to be calculated: the analogy here is that the expected value of verifying a component is proportional to a measure of the criticality of the component multiplied by the probability of it having an exploitable vulnerability.

The calculation is complicated by an attacker's ability to turn what appear to be uncritical components into critical ones. An exploit in a critical component that allows the attacker to reach another component that was assumed to be unreachable may allow additional attacks to be launched. No formula exists to calculate what level of verification should be applied to which components. But we can provide some hints.

First, like in a house, the foundation is critical. The foundation for a software system is the operating system. So it pays to use a high-assurance operating system; for example, seL4 is a formally verified microkernel that is guaranteed to be correct, with respect to its specification and free from undefined behavior [13].

Beyond the foundation and the interfaces, the Pareto Principle applies: in one internal study, Microsoft found that 80% of exploits come from 20% of the bugs [14]. I conjecture the results hold more broadly—i.e., that software vulnerabilities follow a Pareto power law distribution—and furthermore, that the distribution holds for *classes* of vulnerabilities. Recall in Figure 1 that nearly all vulnerabilities exploited in the automotive hacking work were boring buffer overflows.

If the conjecture holds, then the extent to which high-probability-of-exploit classes of vulnerabilities can be eliminated dramatically improves security. I discussed in Section II that buffer overflows (and memory safety errors, in general) can be eliminated.

Finally, do the easy stuff first. Following coding standards, performing unit, regression, and fuzz testing, requiring all compiler warnings be enabled *and* eliminated, and using multiple commercial static analysis tools must be part of the standard engineering practices for building high-assurance cyber-physical systems. While these practices require no specialized expertise in verification or security, they are often ignored, even for life-critical systems [15].

Hint: system verification is a probabilistic game; tilt the odds in your favor.

VI. The Mythical Verification Month

"People don't help much."

Functionality or correctness—it seems a team must choose one. How do you motivate a team to build a functional and correct system in a limited time? The HACMS program had flight demonstration and red-team security review every 18 months. The demonstration and red team assessment were equally important, and because they happened simultaneously, both had to be addressed in the same delivered system.

In the previous sections I have addressed technical aspects of building high-assurance cyber-physical systems. Just as important, however, are the social and cultural aspects. The designers and implementors fight a multi-front battle: building the required functionality and performance while satisfying security, safety, or reliability constraints, all while meeting cost

and scheduling objectives. Balancing these concerns requires the right incentives and culture to meet those incentives.

A prototypical example is the Space Shuttle software group that builds software with an historical error rate of around two errors per million lines of code [16]. They were successful largely through culture alone, without the use of formal verification techniques that were not available in the 90s.

Brooks' *The Mythical Man-Month* famously addresses social and management aspects of building computing systems, and the advice is particularly relevant to building high assurance systems. Brook's law states that "adding manpower to a late software project makes it later" [17]. I propose a corollary, that adding people to a project with low-quality software lowers its quality, *even if those additional people are working on quality assurance*. Additional people working on verification and validation cause the same problems as additional engineers added to a late project: additional communication overhead, additional ramp up time, and working on indivisible tasks, making it very difficult to improve code quality by simply increasing the number of engineers or quality assurance tools.

In summary, Nancy Leveson gives prescient advice from 20 years ago in her book, *Safeware*:

> One obvious lesson is that most accidents are not the result of unknown scientific principles but rather of a failure to apply well-known, standard engineering practices. A second lesson is that accidents will not be prevented by technological fixes alone, but will require control of all aspects of the development and operation of the system [18].

Hint: high-assurance systems require a high-assurance culture.

VII. Remaining Pains

In the following, I mention issues for which I have no hints to give; but are nonetheless critical.

The system configuration and build trust problem is usually ignored, even in high-assurance systems. `GNU Make` is 25k lines of C, with complex built-in and user-defined rules. It is also the defacto build system for embedded systems. Build system maintenance is notoriously difficult [19]. While `GNU Make` is pervasive, well-maintained, and useful, having a large complex build system wrapped around verified software is a bit like putting a skyscraper on quicksand.

In the fault-tolerance community we say that "time turns the improbable into the inevitable".[3] Your system will eventually fail. SMACCMPilot is not a fault-tolerant system, and how to combine security and fault-tolerance is currently a nascent art.

Just as there is no silver bullet in software engineering specifically, there is no silver bullet in high-assurance CPS engineering specifically. Each hint presented is a piece of the puzzle, and taken together, they have a multiplicative effect. High-assurance software is feasible—I have described the Space Shuttle software development process in Section VI. The limiting factor is cost. My hypothesis is that these hints can result in high-assurance CPS at a lower cost than in traditional high-assurance domains. I hope one day we can empirically test the hypothesis.

[3] Attribution unknown.

Acknowledgements

This work is supported in part by AFRL contract FA8750-12-9-0169. All opinions expressed herein are the author's alone. I have been inspired by conversations with my collaborators at Rockwell Collins, Boeing, Data61 (formally NICTA), and University of Minnesota, and my colleagues at Galois.

References

[1] B. W. Lampson, "Hints for computer system design," in *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, ser. SOSP '83. ACM, 1983, pp. 33–48.

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

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

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

[5] T. Elliott, L. Pike, S. Winwood, P. Hickey, J. Bielman, J. Sharp, E. Seidel, and J. Launchbury, "Guilt free Ivory," in *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, ser. Haskell 2015. ACM, 2015, pp. 189–200.

[6] J. P. Anderson, "Computer security technology planning study," Deputy For Command and Management Systems HQ Electronic Systems Division (AFSC), Tech. Rep. ESD-TR-73-51, Vol. II, 1972.

[7] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Trans. Amer. Math. Soc.*, 1953.

[8] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: A declarative language for real-time programming," in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '87. ACM, 1987, pp. 178–188.

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

[10] L. Sassaman, M. L. Patterson, S. Bratus, , and A. Shubina, "The halting problems of network stack insecurity," in *;login:*, vol. 36, 2011.

[11] M. Jones, "What really happened on Mars?" May 2016, available at http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html.

[12] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. ACM*, vol. 17, no. 10, pp. 549–557, Oct. 1974.

[13] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. ACM, 2009, pp. 207–220.

[14] P. Rooney, "Microsoft's CEO: 80-20 rule applies to bugs, not just features," Online article, Oct. 2002, available at http://www.crn.com/news/security/18821726/microsofts-ceo-80-20-rule-applies-to-bugs-not-just-features.htm.

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

[16] C. Fishman, "They write the right stuff," *Fast Company*, 1996, http://www.fastcompany.com/28121/they-write-right-stuff.

[17] F. P. Brooks, Jr., *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1995.

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

[19] P. Miller, "Recursive make considered harmful," in *AUUGN Journal of AUUG Inc.*, vol. 19, 1997, pp. 14–25.