



Requirements and Architectures for Secure Vehicles

Michael W. Whalen, Darren Cofer, and Andrew Gacek



DO YOU TRUST the software in your vehicle? Recent exploits have let hackers remotely control aspects such as brakes and steering, perform surveillance and eavesdropping, and even remotely steal a car. It's speculated that Iran landed a US stealth drone at an Iranian airfield through a GPS spoofing attack. Recent research in self-driving cars and multivehicle coordination requires ever-more software that could be used to launch cyberattacks.

In DARPA's High-Assurance Cyber Military Systems (HACMS) project, research teams are investigating how to construct complex networked-vehicle software securely. An *air team* builds a software stack for unmanned aerial vehicles (UAVs), and a *ground team* investigates software for automobiles and ground-based robots. These teams are paired with a *red team* of professional penetration testers to assess the software's security vulnerabilities. The red team can access all software, design documentation, models, meeting documentation, analysis results, and system binaries produced by the other teams.

To build our air-team software securely enough to repel red-team attacks, we needed an approach that was rigorous, flexible, and compositional, to let us focus on important security concerns at several abstraction levels. As in com-

mercial and military development, our UAVs must incorporate a significant amount of third-party software. We also expect that our UAVs could be networked to construct systems of systems whose purpose might differ considerably from the UAV system's original intent. So, we must be able to reason about requirements at various abstraction levels. Indeed, whether you consider a statement to be a requirement or design decision depends on the abstraction level on which you focus (see Figure 1).

Setting Requirements (and Their Limits)

To define meaningful requirements, we made two main assumptions about the system and potential attackers. These assumptions are essentially limits on what we can prove about the system's security.

The first assumption relates to a UAV's intended functionality and its controllability from the associated ground station. We assume that an authorized user has the authority to issue any command to the UAV, including commands that would crash or otherwise destroy it. We don't wish to limit a priori what a legitimate user may choose to do with the UAV, so we assume that all commands sent by the authorized user are legitimate. We'll only need to model whether a message

(and the command it carries) is well formed. If an attacker can co-opt an authorized user's identity, no mitigation is possible.

The second assumption relates to using wireless communication. Because we can't realistically limit access to the radio spectrum, attackers will always be able to launch a denial-of-service (DoS) attack, by either jamming the physical link or overwhelming the UAV receiver with well-formed messages (even if they fail authorization). This means we can't provide absolute guarantees about reception and execution of commands from authorized users. However, we can require the UAV to reject any commands lacking authorization. We can also require the UAV to execute commands from authorized users in a timely fashion, assuming there's no DoS attack on the radio link. And, when a DoS attack is detected, our requirements can specify what actions the UAV should take to keep itself safe or avoid compromising its mission (if possible).

To construct the requirements, we followed an approach similar to that described in last issue's column, which employed Security Cards.² We knew quite a bit about our immediate adversary, the red team: they had strong technical skills and essentially unlimited knowledge about the system. So, we focused on a variety of known concrete attacks drawn from the Common Attack Pattern Enumeration and Classification list (<http://capec.mitre.org>). First, we ensured generic security principles such as user identification and authorization, secure network access and communication, secure storage, content security, and availability. From those principles, we created system-level

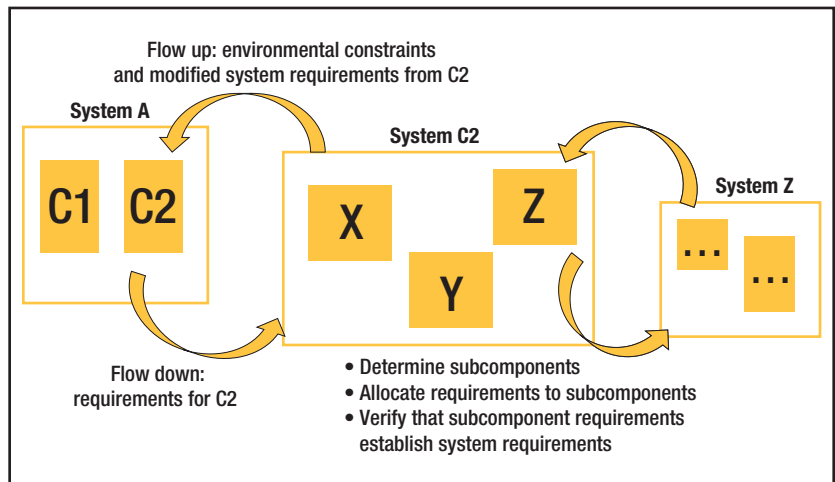


FIGURE 1. The interplay between requirements and architecture.¹ Whether you consider a statement to be a requirement or design decision depends on the abstraction level on which you focus.

security requirements for the UAV, including these:

- The UAV executes only unmodified commands from the ground station.
- If an air-ground communication link fails (or is eliminated through a DoS attack), the UAV executes its no-communication behavior.

From the system requirements, additional requirements were levied on the data link, OS, maintenance procedures, and fault handling, as well as on other system aspects.

Eliminating Weaknesses

Even with good requirements, preventing attacks is difficult; new attack methods are regularly discovered. So, we also focused on common software weaknesses that lead to security problems. The Common Weakness Enumeration website (<http://cwe.mitre.org>) maintains a large list of such weaknesses. Therefore, we approached the problem bottom-up, eliminating common

weaknesses known to be important to many attacks, such as those related to authentication and authorization, system partitioning, maintenance, OS boot and configuration, overflow or underflow, encryption, and memory safety.

Some of these weaknesses depend considerably on the system architecture. We modeled the system architecture in AADL (Architecture Analysis & Design Language)³ and used this model to reason about system vulnerabilities. We then constructed tools to build system images directly from the model. To ensure strict enforcement of the architectural partitioning, we used the seL4 microkernel from Data 61, which has a rigorous proof of correctness.⁴

Other weaknesses can be eliminated by the programming language. For example, Ivory, developed by Galois Inc., is an efficient domain-specific programming language that guarantees the absence of certain classes of memory errors. It also provides significant integration with

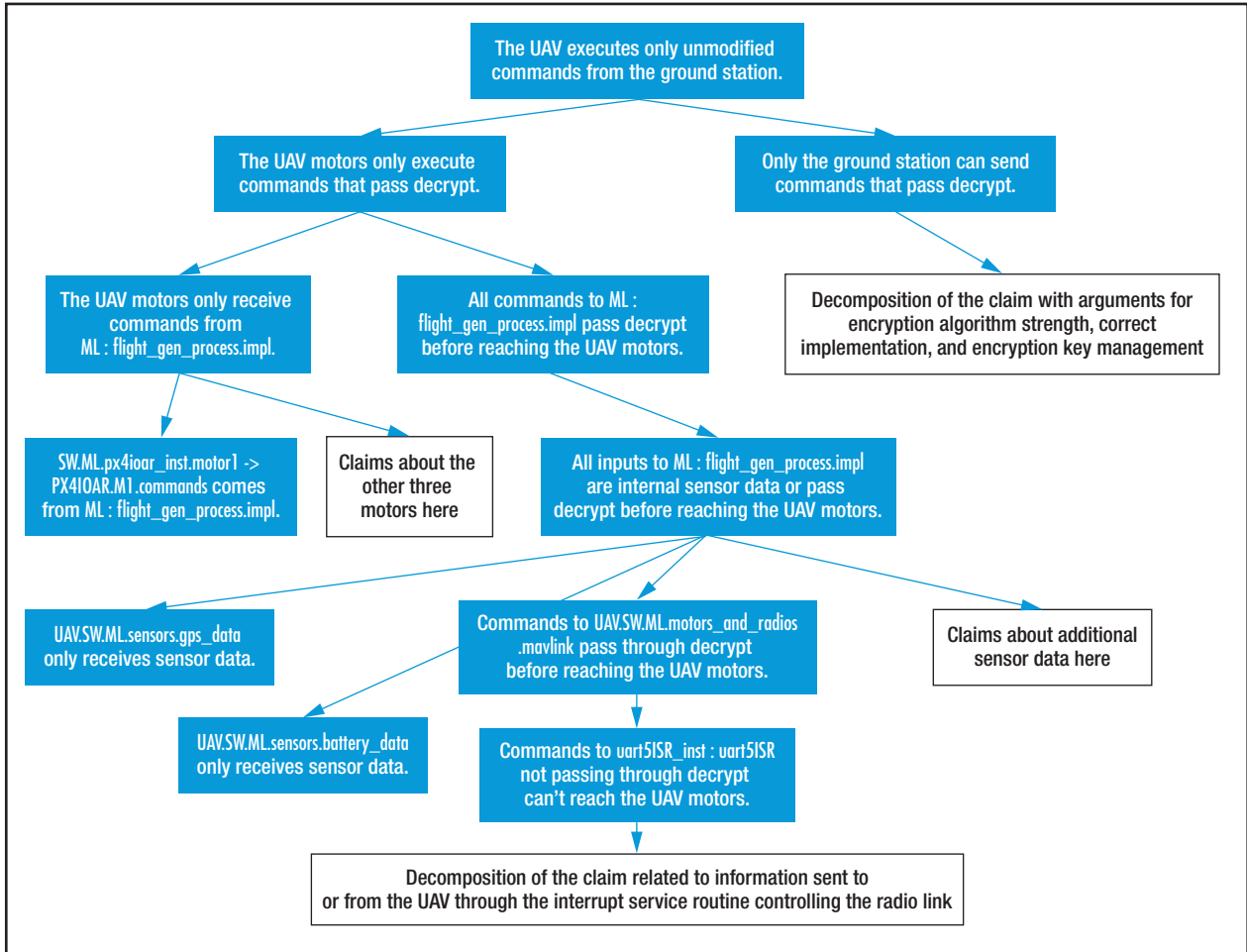


FIGURE 2. A portion of the automatically generated assurance case tree for an unmanned air vehicle (UAV) for the requirement, “The UAV executes only unmodified commands from the ground station.”

model-checking tools to check for underflow or overflow, as well as cryptographic libraries.

Reasoning about Security and Composition

Assume we can define what it means for a component to be secure and can define the property $\text{Secure}(A)$. It might be tempting to say that when we assemble components to form a larger system, we get

$$\text{Secure}(A) \wedge \text{Secure}(B) \Rightarrow \text{Secure}(A \oplus B),$$

for some (as yet undefined) composition operator \oplus . Such a result certainly isn’t true in general, although it might hold for some specific properties and types of composition. For example, if we’re concerned with a system’s memory safety, it might be sufficient to demonstrate the memory safety of all that system’s processes:

$$\text{MemSafe}(A) \wedge \text{MemSafe}(B) \wedge \text{MemSafe}(C) \Rightarrow \text{MemSafe}(\text{System}(A, B, C)).$$

What’s more often the case in

architecture-based composition is that we must look at different properties at the component level rather than the system level. In this way, composition is more like a set of lemmas used in an argument to complete a theorem’s proof. Each lemma might correspond to a particular component property, a channel connecting components, or an attacker. That is,

$$\text{Lem1}(A) \wedge \text{Lem2}(\text{Chan}) \wedge \text{Lem3}(B) \wedge \text{Lem4}(\text{Attack}) \Rightarrow \text{Secure}(A \oplus B).$$

Using such reasoning, we devel-

oped an assurance case for our system's primary high-level property: "The UAV executes only unmodified commands from the ground station." Figure 2 shows a portion of the assurance case. At the top of the case, we separated this claim into two parts on the basis of the encryption setup:


- The UAV motors only execute commands that pass decrypt.
- Only the ground station can send commands that pass decrypt.

We broke down the second part into claims about our encryption protocol's strength and the keys' secrecy. The first part required a more detailed analysis of our software's architecture, including dataflow paths and memory protection.

We expressed our assurance case's general structure using logical expressions in the Resolute language.⁵ Resolute lets you embed assurance case claims and rules in a system's architectural model. The Resolute engine evaluated these rules over our architecture to generate a concrete assurance case. This allowed our assurance case to adapt automatically as the model was updated and changed. Some changes could break the assurance case—for example, adding a non-memory-safe component to a nonpartitioning real-time OS; Resolute would flag such issues. We could then fix this—for example, by hosting the component in seL4 or ensuring that the component was memory-safe by compiling from Ivory. For either fix, Resolute would automatically construct the corresponding assurance case.

In addition, several portions of the architecture were assured through proof. The proof of partitioning and correct OS behavior in

seL4⁴ provided an ironclad foundation for building our UAV. Proofs of the Ivory type system ensured that all Ivory programs that compiled were memory-safe, which removed large classes of attacks.

The HACMS project comprises three 18-month phases. Near each phase's end, the red team receives a demonstration vehicle and software for penetration testing. In phases 1 and 2, the air team's UAVs successfully resisted all red-team attacks. In phase 1, attacks were possible only through the communications link between the ground station and UAV. In phase 2, we provided root access to a Linux partition that controlled a camera used for vehicle tracking and demonstrated that attacks launched from this partition didn't affect the UAV's flight-worthiness. In phase 3, we're adding capabilities to our UAVs, such as secure geofencing to ensure they avoid certain no-fly zones. We're also pursuing technology transfer of our tools and techniques with automotive and aerospace companies. Our experience shows that careful attention to requirements and system architecture, along with formally verified approaches that remove known security weaknesses (using Ivory and seL4), can lead to vehicles that can withstand attacks from even sophisticated attackers with access to vehicle design data. 

Acknowledgments

This research was funded partly by DARPA and the Air Force Research Laboratory under the High-Assurance Cyber Military Systems project (contract FA8750-12-9-0179) and by NASA under contract NNA13AA21C.

References

1. M. Whalen et al., "Your 'What' is My 'How': Iteration and Hierarchy in System Design," *IEEE Software*, vol. 30, no. 2, 2013, pp. 54–60.
2. J. Cleland-Huang et al., "Keeping Ahead of Our Adversaries," *IEEE Software*, vol. 33, no. 3, 2016, pp. 24–28.
3. P.H. Feiler and D.P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st ed., Addison-Wesley Professional, 2012.
4. G. Klein et al., "seL4: Formal Verification of an OS Kernel," *Proc. ACM Symp. SIGOPS 22nd Symp. Operating Systems Principles*, 2009, pp. 207–220.
5. A. Gacek et al., "Resolute: An Assurance Case Language for Architecture Models," *Proc. 2014 ACM SIGAda Ann. Conf. High Integrity Language Technology (HILT 14)*, 2014, pp. 19–28.

MICHAEL W. WHALEN is the director of the University of Minnesota Software Engineering Center. Contact him at mike.whalen@gmail.com.

DARREN COFER is a Fellow at the Rockwell Collins Advanced Technology Center. Contact him at darren.cofer@rockwellcollins.com.

ANDREW GACEK is an industrial logician at the Rockwell Collins Advanced Technology Center. Contact him at andrew.gacek@gmail.com.



See www.computer.org/software-multimedia for multimedia content related to this article.