A Formal Approach to Constructing Secure Air Vehicle Software *

Darren Cofer[1], Andrew Gacek[1], John Backes[1], Michael W. Whalen[2], Lee Pike[3], Adam Foltzer[3], Michal Podhradsky[3], Gerwin Klein[4], Ihor Kuz[4], June Andronick[4], Gernot Heiser[4], and Douglas Stuart[5]

[1] Rockwell Collins Advanced Technology Center
[2] University of Minnesota, Dept. of Computer Science
[3] Galois, Inc.
[4] Data61, CSIRO and University of New South Wales
[5] Boeing Research and Technology

Abstract. Current approaches to cybersecurity for vehicle software rely on patching systems after a vulnerability is discovered. What is needed is a clean-slate, mathematically-based approach for building secure software. Our team has developed new tools based on *formal methods* for building software for unmanned air vehicles (UAVs) that is provably secure against many classes of cyber-attack.

1 Introduction

Embedded systems form a ubiquitous and networked computing substrate that underlies much of society. Modern aircraft and automobiles are complex safety-critical systems in which software is integral to the vehicle control and functionality. Like other embedded systems, vehicles are now networked for a variety of reasons, including the ability to conveniently access diagnostic information, perform software updates, provide innovative features, lower costs, and improve ease of use.

However, researchers (and hackers) have shown that all kinds of networked embedded systems are vulnerable to remote cyber-attack. Researchers at University of Washington and University of California San Diego have demonstrated the ability to completely control an unmodified automobile from a remote location [1]. Security researchers Charlie Miller and Chris Valasek have recently extended this work. Other researchers [2], [3] have been probing for vulnerabilities in the communication and avionics systems of commercial aircraft, though with questionable success. The consequences of a successful cyber-attack against these systems include loss of life or denial of military capabilities, above and beyond the compromise of classified information.

The traditional approach to cybersecurity is reactive, responding to cyber-attacks after they occur by identifying a vulnerability and developing a software

* This work was funded by DARPA contract FA8750-12-9-0179.

patch to eliminate that specific vulnerability. This is a cycle that repeats itself with each newfound vulnerability. Even virus-scanning software cannot keep up with the pace of newly created malware, and in fact, often introduces new vulnerabilities that can be exploited. The situation is even worse for embedded software because it is often difficult to patch due to logical issues or certification constraints.

The High-Assurance Cyber Military Systems (HACMS) research program was established to create technology for the construction of cyber-physical systems that would be resilient against cyber-attacks. The program focused on vehicle control systems because of their complexity, criticality, and significance for the military and civilian worlds. The goal of our research was to break the cycle of "patch and pray" by preventing security vulnerabilities from being introduced during the development process. Achieving this goal requires a fundamentally different approach from what the software community has pursued to date. We have adopted a clean-slate, formal methods-based approach to enable semi-automated code synthesis from executable, composable, formal specifications which are subject to analytic verification.

To assess the security of the software produced, we worked with a *Red Team* of professional penetration testers who evaluated our software and attempted to identiy vulnerabilities. The Red Team had access to all design documentation, models, analysis results, source code and binaries. Throughout the project we engaged the Red Team as "friendly adversaries" who would assess systems and identify any issues discovered so that our systems could be improved in the next development iteration.

Our project in the HACMS program, *Secure Mathematically-Assured Composition of Control Models (SMACCM)*, brings together four main concepts based on formal methods: (1) modeling the system architecture and formal verification of its key security and safety properties, (2) synthesis of software components using languages that guarantee important security properties, (3) use of a formally verified microkernel to guarantee enforcement of communication and separation constraints specified in the architecture, and (4) automatically building the final system from the verified architecture model and component specifications.

To show that this approach is both practical and effective, we applied it to two unmanned air vehicles (UAVs). We first developed the technologies on a modified commercial quadcopter called the *SMACCMcopter*. We then applied the same technologies to Boeing's *Unmanned Little Bird* (ULB), a full-sized optionally-piloted helicopter capable of autonomous flight. Successful flight demonstrations and security evaluations by the Red Team show that our approach can be used to build real systems that are resilient against cyber-attacks.

2 Requirements

To define meaningful security requirements, we started from two assumptions about the system and potential attackers. First, 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. It would be a mistake to a priori limit what a legitimate user may choose to do with a military UAV, so we must assume that all commands sent by an authorized user are legitimate. Thus, the primary focus of our attention is on whether messages (and their associated commands) are well-formed, and whether the encryption that we are using is sufficient to distinguished well-formed from malformed messages. If an attacker can co-opt an authorized user's identity, no straightforward mitigation is possible.

The second assumption relates to the wireless communication. Because we cannot 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 it is not possible to 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 no DoS attack on the radio link. In addition, 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 requirements, 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 security requirements for our UAVs. For example:

- The UAV executes only well-formed commands from the ground station.
- If an air-ground communication link fails, the UAV will execute its no-communication behavior.

We also 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, boot and configuration, overflow or underflow, encryption, and memory safety. The Common Weakness Enumeration website (http://cwe.mitre.org) maintains a large list of such weaknesses.

3 Approach

In this section we present an overview of the four main technologies developed in the project and how they have been integrated into a development process to produce systems that are functionally correct and free from security vulnerabilities. Each technology provides the basis for one of four key elements of *architecture-driven assurance*.

The architecture model is correct: The architecture model specifies the overall organization of the system and defines the interfaces for each subsystem and component, how they interact, and what data they share. We

verify both structural or behavioral properties of the model to demonstrate security. Behavioral properties are specified as formal assume-guarantee contracts.

The components are correct: We must also establish that the components specified in the architecture have been implemented correctly. This means that they must satisfy their requirements as specified in behavioral contracts and that they must be free from vulnerabilities that could be exploited by cyber-attackers.

The system execution semantics matches the model: The architecture model makes both explicit and implicit statements about how the system should execute: execution times and periods for tasks, bindings for threads and processes to CPUs, connections between components and their routing on communication busses. In addition, if there are not connections defined between components, then no data should flow between these components.

The system implementation corresponds to the model: We must also have confidence that the system implementation preserves the properties that have been established for the architecture model and components. We automatically generate all of the code and configuration data needed to build the system directly from the architecture and component models.

3.1 Analyzable Architecture

Developers must have high confidence that the system they eventually build accurately reflects the characteristics of the system design that they reason about. Our tools accomplish this by:

- Allowing developers to model the system that they intend to build in a language with clear syntax and semantics
- Analyzing this model to verify that it meets user defined specifications
- Generating the software that runs on the target platform directly from this model

The Architecture Analysis and Design Language (AADL) has been developed to capture the important design concepts in real-time distributed embedded systems [4]. The AADL language can capture both the hardware and software architecture in a hierarchical format. It provides hardware component models including processors, buses, memories, and I/O devices, and software component models including threads, processes, and subprograms. Interfaces for these components and data flows between components can also be defined. The language offers a high degree of flexibility in terms of architecture and component detail. This supports incremental development where the architecture is refined to increasing levels of detail and where components can be refined with additional details over time.

In AADL, the architectural model includes component interfaces, interconnections, and execution characteristics, but not their implementations. Component implementations are described separately using model-based specification

languages or traditional programming languages which are included by reference in the architecture model. This separation of implementation and architecture is an important factor in achieving scalability for the analysis tools that we have developed.

We have developed two different analysis tools to reason about AADL models. The *Assume-Guarantee Reasoning Environment (AGREE)* is a compositional verification tool that proves behavioral properties about AADL models using modern Satisfiability Modulo Theories (SMT)-based model checkers. The second tool, *Resolute*, generates assurance cases from information embedded in the AADL models. Resolute allows us to construct arguments about properties that are more difficult to formalize, and to integrate heterogeneous sources of evidence about the system.

The Assume Guarantee Reasoning Environment AGREE is used to reason about past-time temporal logic behavioral contracts in AADL architectural components [5]. These contracts consists of assumptions about the component environment and guarantees about how the component state evolves over time . A contract specifies precisely the information that is needed to reason about the components interaction with other parts of the system. Furthermore, the contract mechanism supports a hierarchical decomposition of the verification process that follows the natural hierarchy in the system model.

Given a top-level component that is composed of several subcomponents, AGREE attempts to prove that the top-level component contract holds, given the top-level contract assumptions and assuming that the contracts of its subcomponents are true. The reasoning is performed using a state-of-the-art inductive model checker called jKind [6]. This decomposition can be performed for any number of architectural layers, allowing compositional reasoning across a large-scale system architecture. The proof rests on "leaf-level" contracts over individual threads or processes, which must be discharged by other means (such as model checking or coverage-based testing). If AGREE is unable to produce a proof, then it produces a counterexample that illustrates a scenario in which the system-level contract guarantee does not hold, given the system-level assumptions and subcomponent contracts.

As an example, we used AGREE to verify the correct implementation in the ULB of a distributed protocol (STANAG 4586) for controlling interactions among multiple ground stations and UAVs. STANAG 4586 defines messages that request various levels of control over the UAV, such as setting new waypoints or controlling an onboard camera. These messages require different authoriy, called *levels of interoperability (LOI)*, to interact with the vehicle. It is crucial that the vehicle not act upon messages sent by a ground station with an inadequate LOI. Likewise it is important that a UAV only grant an LOI to a ground station that is appropriate based on the current state of the vehicle and the permissions decided upon at the beginning of the mission. We used AGREE tool to model and verify these properties.

Resolute Traditional assurance cases are informal arguments for the correctness of a system. Each claim in the argument is supported by other sub-arguments or evidence, resulting in a tree shape. Resolute formalizes and extends this notion. First, the dependency of each argument on its sub-arguments and evidence is formalized into rules. Second, these rules can be parameterized by the architecture of the system (e.g. iterating over all components). Finally, Resolute instantiates these rules for a particular architecture using a Datalog-style proof search algorithm [7]. Resolute assurance cases are automatically updated as the architecture evolves, and they never fall out of sync with architecture.

Consider an assurance case for the claim "The UAV executes only unmodified commands from the ground station." We can decompose this claim into two arguments: one about the correctness of our encryption algorithm and one about the data-flows between the Decrypt component and the eventual execution of commands. The latter property is particular interesting for Resolute since it relies on the currect architecture of the system. We formalize it with a recursive rule which describes when a component receives properly decrypted messages. Resolute traverses the the architecture to track how messages move through the system and compute the validity of the claim.

3.2 Correct Components

The next aspect of our approach requires that software components specified in the architecture model, such as threads or functions, be correctly implemented. C or C++ are still the most common languages for embedded system development given the low-level control they provide in terms of memory usage and timing behavior. Unfortunately, these languages provide little support for creating high assurance software. Used on their own, they are not memory safe and difficult to analyze.

To address this problem, our team developed an embedded domain-specific language (EDSL) called *Ivory*. This language was used to re-implement all of the flight control functions in the SMACCMcopter research vehicle and critical control and communication functions in the ULB.

Ivory [8] follows in the footsteps of other "safe C" programming languages, like Cyclone, BitC, and Rust—languages that avoid many of the pitfalls of C, particularly related to memory safety and undefined behavior, while being suitable for writing low-level code (e.g., device drivers), and having minimal runtime systems. Our main motivation for not using those languages is our desire for an EDSL providing convenient, Turing-complete, type-safe macro-language (Haskell) to improve productivity.

Ivory is particularly designed for safety-critical embedded programming. Such a language should guarantee memory safety, prevent most undefined behaviors, and provide integrated testing and verification tools. Typical C coding conventions for safe embedded systems, such as those in use at NASA's Jet Propulsion Laboratory [9], are enforced by Ivory's type system. In line with these conventions, Ivory has been built with some limitations to simplify generating safe C programs. Ivory does not support heap-based dynamic memory allocation (but

global variables can be defined). Arrays are fixed-length. There is no pointer arithmetic. Pointers are non-nullable. Union types are not supported. Unsafe casts are not supported: casts must be to a strictly more expressive type (e.g., from an unsigned 8-bit integer to an unsigned 16-bit integer) or a default value must be provided for when the cast is not valid. The most common unsafe C cast is not possible: no void-pointer type exists in Ivory.

In practice, Ivory has proven to be a tremendously productive language, both in spite of and due to these restrictions and limitations. Ivory programmers get the full power of using Haskell as a macro system, while being reassured by the type system that their programs are safe. For example, the extended Kalman filters used for state estimation on the SMACCMcopter were generated from a high-level description of the algorithm in terms of linear algebra operations, but produced safe C code nearly identical to hand-unrolled loops. Meanwhile, the very lowest levels of detail in SMACCMcopter's board support package were developed using distinct types for register flags and addresses, eliminating the mismatches that are common when dealing with bit masks and hardware addresses directly.

3.3 Execution Semantics and Operating System

Once we are satisfied that the architecture has been correctly specified and the software components correctly implemented, the correct execution of the components, isolation between components, and enforced communication between components must be guaranteed. This is ensured by the the underlying operating system (OS).

Each of our UAVs includes two computers: A flight control computer for hard real-time control tasks, and a mission computer for communicating with the outside world (the ground station, in particular) and hosting onboard payloads such as a video camera. These computers have very different requirements and run different operating systems.

The OS used on the mission computers of both of our UAVs is the seL4 microkernel. The seL4 microkernel builds on the strengths of the L4 microkernel architecture, such as small size, high performance, and policy freedom, and extends it with a built-in capability model, which provides a mechanism to enforce security guarantees at the operating system and application levels. The seL4 microkernel has undergone extensive formal verification, from full functional correctness down to binary level, to strong high-level security properties including confidentiality and integrity [10]. This means that seL4's executable implementation is formally proved correct relative to its specification using mathematical machine-checked proofs in the Isabelle/HOL theorem prover [11]. Its security properties, also proved in Isabelle/HOL, imply that isolation is enforced; i.e. that seL4 does enforce the controlled communication defined in the component configuration of the architectural specification. The isolation and controlled communication enforcement is the key to showing that the AADL architecture model is properly implemented.

On the flight control computers, the focus is on ensuring timely execution and scheduling of flight tasks, leading to use of a real-time operating system (RTOS). On the SMACCMcopter we have used eChronos, a formally verified RTOS developed by Data61 that runs on highly resource-constrained hardware.

On the ULB we have used the VxWorks RTOS. Use of this commercial RTOS was necessary because of the particular flight computer hardward in the ULB. While not optimal, use of an RTOS without the assurance provided by formal verification was deemed acceptable since the flight computer is isolated from contact with the outside world by the mission computer running seL4.

3.4 Trusted Build

Finally, we must ensure that the guarantees designed in to the architectural models, software components, and OS are preserved in the actual system implementation. To ensure conformance, we built tools to automatically generate the system image directly from the architectural model, software components, and OS code. For both vehicles, the AADL architecture model was detailed enough to support generation of "glue code" and all configuration information needed to construct a system image that could be loaded directly onto the target platform.

We developed the *Trusted Build* (TB) tool to generate system images from AADL models. From AADL models, TB can generate the OS configuration information, process/thread priorities and scheduling information, and all process/thread communication primitives. In fact, it is also possible to automatically generate communication primitives between operating systems, as happens with virtual machines. TB allowed single-source models to target the VxWorks, eChronos, seL4, or Linux operating systems, depending on the needs of the specific platform. The final system images generated for both vehicles were generated directly from the AADL architecture descriptions using TB. While the majority of the TB tool was not formally verified, the communications primitives used for IPC in seL4 were verified using Isabelle/HOL.

4 Application and Demonstration

We demonstrated our approach on two different UAVs: the SMACCMcopter quadcopter, and the Boeing Unmanned Little Bird helicopter (1). This section describes our experiences with both platforms.

4.1 SMACCMcopter Demonstration

The SMACCMcopter was developed as an open experimentation platform that would be available for use by researchers without restriction. It is based on commercially available hardware components and open source software. It mimics the architecture and features of the ULB in a number of ways, and has been a practical way to develop, refine, and test new technologies.

Fig. 1. Demonstration Aircraft: SMACCMcopter and Unmanned Little Bird

The airframe for the SMACCMcopter is the IRIS+ quadcopter produced by 3D Robotics. The IRIS+ uses a Pixhawk flight control computer which runs the hard real-time control software and includes integrated sensors for vehicle acceleration and attitude. A separate mission computer has been mounted on top of the IRIS+ body. The mission computer is based on an ARM Cortex-A15 CPU and communicates with the flight control computer over a CAN bus. It hosts functions for encryption/decryption, the CAN interface to the flight computer, and ground station communication. To demonstrate mixed-security architectures involving commercial software, the camera software represents an untrusted component that runs in a Linux virtual machine (VM) hosted by seL4. It receives video data from the camera, detects and computes bounding boxes for objects of a specified color, and sends video data to the ground station.

All of the SMACCMcopter software was written using the approach described in Section 3. The secure Ivory software components, secure seL4 operating system, and verified AADL software architecture result in a quadcopter design in which most common security vulnerabilities have been eliminated. A simplified diagram of the architecture is shown in Figure 2.

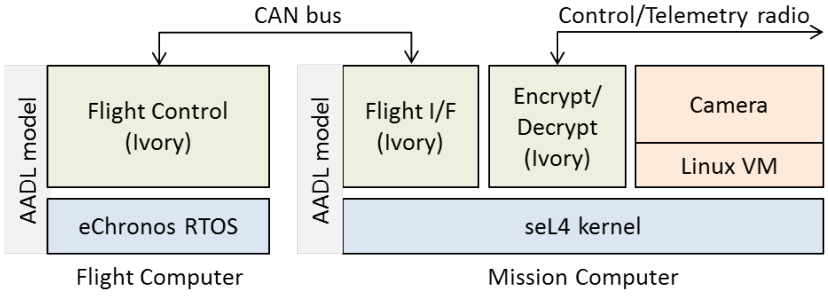

Fig. 2. Simplified software architecture for SMACCMcopter showing verified OS (blue), trusted components (green), and untrusted components (orange)

During the course of the HACMS program, we conducted flight tests to demonstrate the effectiveness of our approach and tools applied to the SMACCMcopter. The demonstration consisted of two scenarios illustrating the difference between an unsecure, unverified version of the SMACCMcopter software and the final secure, verified version of the software. In each scenario, the SMACCMcopter was flown and commanded by the ground control station while a separate team of "attackers" launched cyber-attacks on the vehicle, attempting to take over its telemetry and flight control. In the first scenario, the cyber-attack was shown to be successful, resulting in the attackers gaining complete control of the vehicle. In the second scenario, the formally verified SMACCMcopter was able resist the same attacks and complete its mission unhindered. A video of this demonstration is available at [12].

4.2 Unmanned Little Bird Demonstration

The ULB is an optionally-piloted helicopter based on the H-6, a 32 foot long, 4700 pound rotorcraft. The ULB adds an autonomous capability to the basic H-6. Though the ULB is capable of fully autonomous flight, for flight testing it carries a safety pilot who can disable and override the autonomous functionality.

Like the SMACCMcopter, the ULB avionics includes a flight control computer (FCC) for real-time tasks and a mission computer (called the Vehicle Specific Module, or VSM) for communication with the ground station and managing a video camera payload. The original ULB VSM was implemented in 87K lines of C++ source code, with an executable size of approximately 80 MB, running on Gentoo Linux on an x86 processor. The original ULB FCC was written in 20K lines of C code, with a 2MB executable, using a monolithic cyclic executive running at 50 Hz on a PowerPC platform. During the HACMS program the Boeing ULB program ported the FCC software to VxWorks, which increased the code size to approximately 40K lines. The ULB implements the STANAG 4586 protocol for communication between ground stations and UAVs. The protocol permits any compliant ground station to control any compliant UAV.

Over the course of the three phases of the HACMS program, new technologies were progressively applied to the ULB to create a high-assurance cyber military system. In Phase 1, the VSM architecture was modeled in AADL, and seL4 was added as a hypervisor to host the baseline VMS on its baseline Linux operating system as a guest operating system. In Phase 2, the Ivory language was used to re-implement a portion of the VSM software, along with new authentication and LOI components. A more detailed AADL model of the VSM software architecture was developed and used with the Trusted Build tool to generate code for the VSM. In Phase 3, the FCC software architecture was modeled using AADL, and the outer loop control and input/output components of the FCC were implemented in Ivory. In this case the existing VxWorks RTOS was retained as the operating system. The resulting final ULB HACMS architecture is shown in Figure 3.

Several ULB flight tests were conducted to demonstrate that the vehicle with updated cybersecure software retained all of its original functionality. As

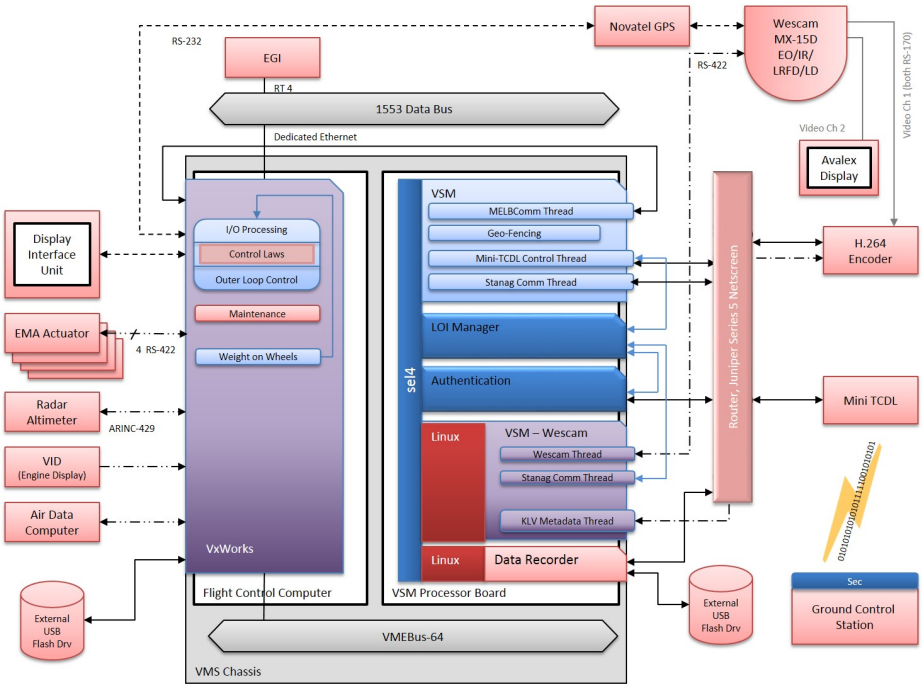

Fig. 3. ULB Final Architecture showing verified OS and trusted components (blue) and unmodified/untrusted components (red)

with the SMACCMcopter, we flew several sorties that included targeted cyber-attacks. In the first attack, a compromised maintenance device was connected to the USB socket on the ULB that normally hosts a USB drive used for the data logging. This device injected a virus which attempts to move from the data logging software to the other VSM software, ultimately causing the payload camera to be inoperative. In the second attack, a simulated *supply chain* attack originating in the camera software attempts to change ULB waypoints and cause it to violate (simulated) airspace restrictions. In the final upgraded version of the ULB software both of these attacks were thwarted.

The technologies described here were applied to the ULB by Boeing engineers (with some support from the technology researchers). Significantly, this included engineers from Boeing Defense Systems, as well as those from Boeing Research and Technology. Together, this represents non-trivial evidence that these technologies are effective in improving system cybersecurity, can do so for real aircraft without compromising the required real-time performance, and are usable by the developers of military systems.

5 Conclusion

Over the course of the HACMS program, a number of formal methods technologies were developed and applied, first to the SMACCMcopter research vehicle, and then to the Boeing Unmanned Little Bird helicopter.

In the beginning of the program, the Red Team performed baseline assessments of both our "stock" Pixhawk-based hobby quadcopter and the original ULB software. In both baselines, the Red Team had little difficulty in attacking the vehicles. The quadcopter was trivially compromised in several ways (e.g. hijack of unencrypted communications, message flooding, and several other issues) and the ULB was compromised within an hour due to configuration and memory issues involving third-party components. Over the three phases of the project, our new technologies and software assumed more and more of the of control of the vehicles until, in Phase 3, they formed the entirety of the SMACCMcopter and the majority of the ULB.

These technologies were successfully demonstrated on both aircraft in flight, including the successful defeat of attacks using a variety of common attack vectors. The SMACCMcopter withstood attacks via a remote data link, while the ULB withstood attacks via a compromised USB device and compromised third-party software for an onboard payload.

After each phase, the Red Team performed a security assessment of the upgraded portions of the vehicle software and performed penetration testing. After Phase 1, their evaluation and penetration testing focused on remote attacks on the vehicles. In later phases this expanded to include attacks launched from non-critical components onboard the vehicles themselves. The Red team assessments did not find any exploitable vulnerabilities in the re-engineered portions of either aircraft.

At the end of the project, Red Team final report concluded: *HACMS technologies have made revolutionary advances in the resilience available to developers of autonomous vehicles. The final vehicles delivered under the HACMS program, even as research prototypes, proved to be resilient against most forms of attack to a degree rarely seen even in hardened, fielded systems. Of all the final, formally verified components assessed under the final phase of the program, no memory corruption failures, mathematical operation faults, or security isolation compromises were identified.*

In this project we have demonstrated the use of formal methods to dramatically improve the cybersecurity of the embedded software in two aircraft. In addition to security assessments, these aircraft underwent flight testing to show that their real-time performance had not been impacted. Furthermore, all of the modification and re-engineering of the ULB software was conducted by Boeing engineers. Thus, the formal methods technologies presented here are both practical and effective in enhancing the cybersecurity of real aircraft.

All of the models, software, and tools developed as part of this project are open source and available in the SMACCM github repository (https://github.com/smaccm/smaccm).

References

1. Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*, 2011.
2. Hugo Teso. Aircraft hacking: Practical aero series, 2013. https://conference.hitb.org/hitbsecconf2013ams/hugo-teso/ (accessed 4-28-2017).
3. Kim Zetter. Feds say that banned researcher commandeered a plane, 2015. https://www.wired.com/2015/05/feds-say-banned-researcher-commandeered-plane/ (accessed 4-28-2017).
4. P. Feiler and D. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language.* Addison-Wesley Professional, 1st edition, 2012.
5. Michael W. Whalen, Andrew Gacek, Darren D. Cofer, Anitha Murugesan, Mats Per Erik Heimdahl, and Sanjai Rayadurgam. Your "what" is my "how": Iteration and hierarchy in system design. *IEEE Software*, 30(2):54–60, 2013.
6. Andrew Gacek. JKind - a Java implementation of the KIND model checker. https://github.com/agacek, 2015.
7. A. Gacek et. al. Resolute: An assurance case language for architecture models. In *HILT 2014*, pages 19–28, New York, NY, USA, 2014. ACM.
8. P. Hickey et. al. Building embedded systems with embedded DSLs (experience report). In *Intl. Conference on Functional Programming (ICFP)*. ACM, 2014.
9. JPL. JPL institutional coding standard for the C programming language. Technical Report JPL DOCID D-60411, Jet Propulsion Laboratory, 2009. Available at `http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf`.
10. G. Klein et. al. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

11. T. Nipkow et. al. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of LNCS. Springer, Heidelberg, 2002.
12. Darren Cofer, Andrew Gacek, John Backes, and Konrad Slind. Video: High-assurnce cyber military systems (hacms), 2017. https://insights.rockwellcollins.com/2017/07/06/video-high-assurance-cyber-military-systems-hacms/ (accessed 3-2-2018).