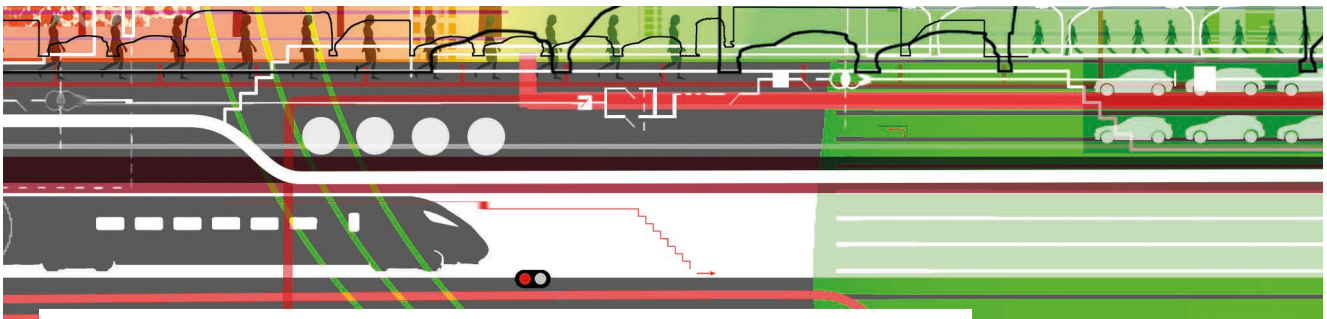

A Formal Approach to Constructing Secure Air Vehicle Software

Darren Cofer, Andrew Gacek, and John Backes, Rockwell Collins Advanced Technology Center
Michael W. Whalen, University of Minnesota
Lee Pike, Adam Foltzer, and Michal Podhradsky, Galois Inc.
Gerwin Klein, Ihor Kuz, June Andronick, and Gernot Heiser, Data61, CSIRO, and University of New South Wales
Douglas Stuart, Boeing Research and Technology

Current approaches to cyberresiliency rely on patching systems after a vulnerability is discovered. What is needed is a clean-slate, mathematically based approach for building secure software. We developed new tools based on formal methods for building software for unmanned air vehicles that is provably secure against cyberattacks.

Researchers (and hackers) have shown that all kinds of networked, embedded systems are vulnerable to remote cyberattack. Researchers at the University of Washington and University of California San Diego 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[2–4] have been probing for vulnerabilities in the communication and avionics systems of commercial aircraft, although with questionable success. Above and beyond the compromise of classified information, the consequences of a successful cyberattack against an aircraft include loss of life or denial of military capabilities.

As part of the High-Assurance Cyber Military Systems (HACMS) research program, our team conducted actual cyberattacks on a military aircraft during flight.[5] Our "before" and "after" attacks demonstrated the effectiveness of technologies developed during the HACMS program to construct air vehicles that are resilient against cyberattacks. Cyberresiliency means that the

Digital Object Identifier 10.1109/MC.2018.2876051
Date of publication: 15 January 2019

system is tolerant to cyberattacks in the same way that safety-critical systems are tolerant to random faults—they recover and continue to execute their mission without interruption.

The traditional approach to cybersecurity is reactive, responding to cyberattacks after they occur by identifying a vulnerability and developing a software 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 HACMS 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 has been pursued by the software community to date. We have adopted a clean-slate, formal-methods-based approach to enable semiautomated code synthesis from executable, composable formal specifications that 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 identify vulnerabilities. The Red Team had access to all design documentation, models, analysis results, source codes, 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. However, the cyberresiliency of our software follows primarily from the formal verification effort, not from the subsequent testing and evaluation.

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, which we have 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 cyberattacks.

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 distinguish well-formed from malformed messages. If an attacker can coopt 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 the 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 (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, and
- if an air–ground communication link fails, the UAV will execute its no-communication behavior.

We also approached the problem from the 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 (cwe.mitre.org) maintains a large list of such weaknesses.

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 and 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 cyberattackers.

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, and connections between components and their routing on communication buses. In addition, if no connections are 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.

Analyzable architecture

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

- allowing developers to model the system they intend to build in a language with clear syntax and semantics,
- analyzing this model to verify that it meets user-defined specifications, and
- 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.[6] The AADL 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 as well as 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 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 that are included by reference in the architecture model. This separation of implementation and architecture is an important factor in achieving scalability for the analytic tools we have developed.

These include two different analytic tools to reason about AADL models. The Assume–Guarantee Reasoning Environment (AGREE)[7] is a compositional verification tool that proves behavioral properties about AADL models using modern Satisfiability Modulo Theories-based model checkers. The second tool, Resolute,[8] 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.

Assume–guarantee reasoning environment. AGREE is used to reason about past-time temporal logic behavioral contracts in AADL architectural components. These contracts consist 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 component's 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. Unlike other compositional reasoning tools (such as OCRA[9]), AGREE is fully integrated with AADL so that the embedded implementation can be automatically generated from the verified system model.

Given a top-level component 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*.[10] 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 authority, 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 grant an LOI only 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 to model and verify these properties.

Resolute. Traditional assurance cases are informal arguments for the correctness of a system, such as the goal-structuring notation.[11] Each claim in the argument is supported by other subarguments or evidence, resulting in a tree structure. Resolute formalizes and extends this notion, allowing assurance cases to be attached to AADL models. First, the dependency of each argument on its subarguments and evidence is formalized into rules. Second, these rules can be parameterized by the system architecture (for example, iterating over all components). Finally, Resolute instantiates these rules for a particular AADL architecture using a Datalog-style proof search algorithm. Resolute assurance cases are automatically updated as the architecture model evolves, and they never fall out of sync with the model. An approach to apply and evolve assurance cases as part of system design[12] is similar to the process we have used with Resolute.

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 dataflows between the Decrypt component and the eventual execution of commands. The latter property is particularly interesting for Resolute because it relies on the architecture of the system. We formalize it with a recursive rule that describes when a component receives properly decrypted messages. Resolute traverses the architecture to track how messages move through the system and compute the validity of the claim.

Correct components

The next aspect of our approach requires that the software components

specified in the architecture model, such as threads or functions, be correctly implemented. C and 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 are difficult to analyze.

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

Ivory[13] 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 (for example, device drivers) and having minimal run-time systems. Our main motivation for not using those languages is our desire for an EDSL that provides a convenient, Turing-complete, type-safe macrolanguage (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,[14] 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 (for example, from an unsigned 8-b integer to an unsigned 16-b integer), or a default value must be provided for instances 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 the SMACCMcopter 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.

THE AADL CAN CAPTURE BOTH THE HARDWARE AND SOFTWARE ARCHITECTURE IN A HIERARCHICAL FORMAT.

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 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 a different OS.

The OS used on the mission computers of both of our UAVs is the seL4 microkernel, which 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 that provides a mechanism to enforce security guarantees at the OS

and application levels. The seL4 microkernel has undergone extensive formal verification, from full functional correctness down to the binary level and then to strong high-level security properties including confidentiality and integrity.[15] This means that seL4's executable implementation is formally proved correct relative to its specification using mathematical, machine-checked proofs in the Isabelle/Higher Order Logic (HOL) theorem prover.[16] Its security properties, also proved in Isabelle/HOL, imply that isolation is enforced; that is, the seL4 does enforce the controlled communication defined in the component configuration of the architectural specification. The isolation and controlled communication enforcement are 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 OS (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 hardware in the ULB. While not optimal, use of an RTOS without the assurance provided by formal verification was deemed acceptable because the flight computer is isolated from contact with the outside world by the mission computer running seL4.

Trusted build

Finally, we must ensure that the guarantees designed into 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 the generation of "glue code" and all configuration information needed to construct a system image that can be loaded directly onto the target platform.

FIGURE 1. The demonstration aircraft: an SMACCMcopter and the Boeing Unmanned Little Bird.

We developed the Trusted Build (TB) tool to generate system images 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 OSs, as happens with virtual machines (VMs). TB allowed single-source models to target the VxWorks, eChronos, seL4, or Linux OSs, depending on the needs of the specific platform. The final system images generated for both vehicles were produced directly from the AADL architecture descriptions using TB. While the majority of the TB tool was not formally verified, the communications primitives used for interprocess communication in seL4 were verified using Isabelle/HOL.

THE OS USED ON THE MISSION COMPUTERS OF BOTH OF OUR UAVs IS THE SEL4 MICROKERNEL.

APPLICATION AND DEMONSTRATION

We demonstrated our approach on two different UAVs: the SMACCMcopter quadcopter and the Boeing ULB helicopter (see Figure 1). This section describes our experiences with both platforms.

SMACCMcopter demonstration

The SMACCMcopter was developed as an open experimentation platform available for use by researchers

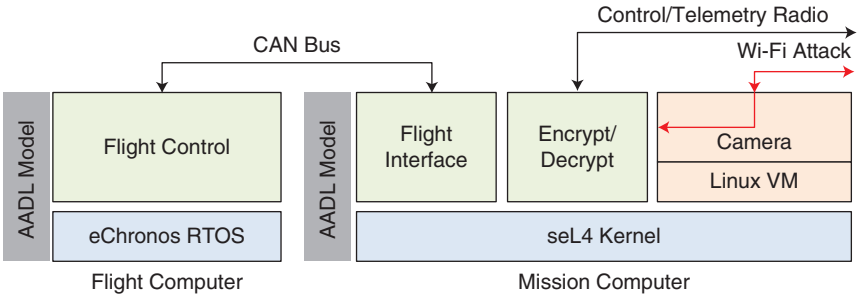

FIGURE 2. The simplified software architecture for the SMACCMcopter showing the verified operating system (OS) (blue), Ivory-synthesized components (green), untrusted components (orange), and Wi-Fi cyberattack (red). AADL: Architecture Analysis and Design Language; VM: virtual machine; RTOS: real-time operating system.

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.

The airframe for the SMACCMcopter is the IRIS+ quadcopter produced by 3D Robotics. The IRIS+ uses a Pixhawk flight-control computer that 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 Controller Area Network (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 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 SMACCMcopter software was written using the approach described earlier. 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.

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 final 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 commanded by the ground control station while a separate team of "attackers" launched cyberattacks on the vehicle, attempting to take over its telemetry and flight control via a Wi-Fi connection to the VM hosting the unverified camera software. In the first scenario, the cyberattack was successful. The attackers were able to remotely access memory containing encryption keys for the control/telemetry radio link and take control of the vehicle. In the second scenario, the formally verified SMACCMcopter was resilient against the same attack and completed its mission unhindered. A video of this demonstration is available online.[17]

Unmanned Little Bird demonstration

The ULB is an optionally piloted helicopter based on the H-6, a 32-ft-long, 4,700-lb rotorcraft. The ULB adds an autonomous capability to the basic H-6. Although 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* (*VSM*)] for communication with the ground station and managing a video camera payload. The original ULB VSM was implemented in 87-K 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 20-K lines of C code, with a 2-MB 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 40-K 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 software running on Linux as a guest OS. In phase 2, the Ivory language was used to reimplement 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 TB 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 OS. A simplified version of the 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 with the SMACCMcopter, we flew several sorties that included targeted cyberattacks. In the first attack, a compromised maintenance device was connected to the USB socket on the ULB, which normally hosts a USB drive used for the data logging. This device injected a virus that attempted to access memory in the other VSM software and disable the payload camera. In the second attack, a simulated supply chain attack originating in the third-party camera software attempted to change the ULB waypoints and cause it to violate (simulated) airspace restrictions. In the final upgraded version of the ULB, both of these attacks were contained by the verified software and system design, allowing the aircraft to continue operation.

The technologies described here were applied to the ULB by Boeing engineers (with some support from the

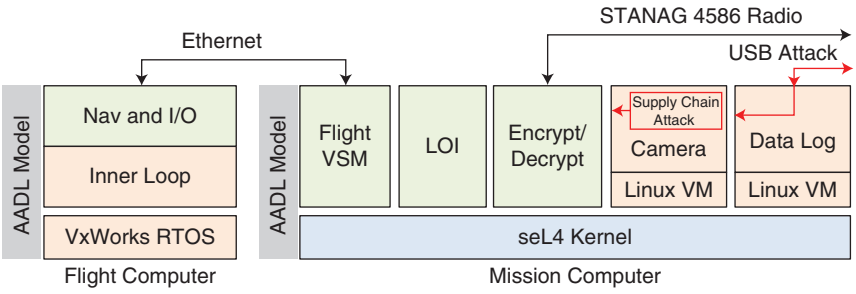

FIGURE 3. Boeing's ULB final architecture showing the verified OS (blue), Ivory-synthesized components (green), unmodified/untrusted components (orange), and two cyberattacks demonstrated (red). VSM: vehicle-specific module.

technology researchers). Significantly, this included engineers from Boeing Defense Systems as well as those from Boeing Research and Technology. Together, this represents nontrivial 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.

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 ULB helicopter.

At the beginning of the program, the Red Team performed baseline assessments of both our unmodified Pixhawk-based hobby quadcopter and the original ULB software. In both baselines, the Red Team had little difficulty attacking the vehicles. The quadcopter was trivially compromised in several ways (for example, 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 during flight, including the successful defeat of attacks based on several of the 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 along with 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 noncritical components onboard the vehicles themselves. The Red Team assessments did not find any exploitable vulnerabilities in the reengineered portions of either aircraft.

ABOUT THE AUTHORS

DARREN COFER is a fellow in the Rockwell Collins Advanced Technology Center. His research interests include formal methods and tools for verification and certification of high-integrity systems. Cofer received a PhD in electrical and computer engineering from the University of Texas at Austin and is a Senior Member of the IEEE. Contact him at cofer@ieee.org.

ANDREW GACEK is a researcher in the Rockwell Collins Advanced Technology Center. His research interests include connecting users with formal verification through tool development and research. Gacek received a PhD in computer science from the University of Minnesota. Contact him at andrew.gacek@gmail.com.

JOHN BACKES is a researcher at the Rockwell Collins Advanced Technology Center. His research interests include surface-mount technology solvers, model checking, and verification of software for embedded systems. Backes received a PhD from the Department of Electrical and Computer Engineering at the University of Minnesota. Contact him at john.backes@gmail.com.

MICHAEL W. WHALEN is the director of the University of Minnesota Software Engineering Center. His research interests involve improving the scalability and usability of model checking and automated test generation. Whalen received a PhD from the University of Minnesota and is a Senior Member of the IEEE. Contact him at mwwhalen@umn.edu.

LEE PIKE is a member of the technical staff at Groq, Inc. Previously, he directed the Cyber-Physical Systems group at Galois Inc., where the research reported herein was completed. His research interests include formal methods, functional programming, and high-assurance systems. Pike received a PhD from Indiana University. Contact him at leepike@gmail.com.

ADAM FOLTZER, previously at Galois Inc., is now a senior software engineer at Fastly, working at the intersection of compilers, performance, and security. Foltzer received an MS in computer science from Indiana University specializing in programming language theory and implementation. Contact him at acfoltzer@acfoltzer.net.

MICHAL PODHRADSKY is a software engineer at Galois Inc. His research is focused on high-assurance cyber-physical systems and, in particular, unmanned aerial vehicles. Podhradsky received a PhD in electrical and computer engineering from Portland State University. Contact him at mpodhradsky@galois.com.

GERWIN KLEIN is a chief research scientist at Data61, CSIRO, and a conjoint professor at the University of New South Wales, Sydney, Australia. His research is on formal software verification (particularly in operating systems), on interactive theorem proving, and in programming languages. Klein received a PhD in computer science from the Technische Universität München. Contact him at gerwin.klein@data61.csiro.au.

IHOR KUZ is a principal research engineer in the Trustworthy Systems group at Data61, CSIRO, and a conjoint associate professor at the University of New South Wales, Sydney, Australia. His research interests are in secure systems, particularly secure operating systems and componentized systems. Kuz received a PhD from Technische Universität Delft and is a Member of the IEEE and ACM. Contact him at ihor.kuz@data61.csiro.au.

JUNE ANDRONICK is a principal research scientist at Data61, CSIRO, and a conjoint associate professor at the University of New South Wales, Sydney, Australia. She is leader of the Trustworthy Systems group, known for formal verification of the seL4 operating system microkernel. Her research interests are in formal verification of concurrent operating system code. Contact her at june.andronick@data61.csiro.au.

GERNOT HEISER is Scientia Professor and John Lions Chair of computer science at the University of New South Wales, Sydney, Australia, as well as a chief research scientist at Data61, CSIRO. His research is on operating systems, especially microkernel-based systems for safety- and security-critical uses, cybersecurity, real-time systems, and architectural support for operating systems. Heiser received a PhD from ETH Zurich and is a Fellow of the IEEE, ACM, and the Australian Academy of Technology and Engineering. Contact him at gernot@unsw.edu.au.

DOUGLAS STUART is a researcher with Boeing Research and Technology. His research interests include cyber-physical systems development, verification, and cybersecurity. Stuart received a PhD from the University of Texas at Austin. Contact him at douglas.a.stuart@boeing.com.

At the end of the project, the 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 reengineering of the ULB software was conducted by Boeing engineers. Thus, the formal methods technologies presented here are both practical and effective in enhancing the cyber-resiliency of real aircraft.

More information, including the final report, models, software, and tools developed as part of the project, is available at loonwerks.com /projects/hacms.html.

ACKNOWLEDGMENTS

This work was funded by DARPA contract FA8750-12-9-0179. The views, opinions, and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the US government.

REFERENCES

1. C. Stephen, D. MCoy, K. Brian, A. Danny, S. Hovav, S. Stefan, K. Karl, C. Alexei, R. Franziska, and K. Tadayoshi, "Comprehensive experimental analyses of automotive attack surfaces," in *Proc. 20th USENIX Security Symp.*, San Francisco, CA, 2011.
2. H. Teso, "Aircraft hacking: Practical aero series 2013," HITB. Accessed on: Aug., 15, 2018. [Online]. Available: https://conference.hitb.org /hitbsecconf2013ams/hugo-teso/
3. K. Zetter. (2015). Feds say that banned researcher commandeered a plane. *Wired*. [Online]. Available: https://www.wired.com/2015/05 /feds-say-banned-researcher-commandeered-plane/
4. R. Santamarta, *Last Call for SATCOM Security.* Las Vegas, NV: Black Hat, 2018.
5. G. Warwick (2017). DARPA blocks cyberattacks on Unmanned Little Bird in flight. Aerospace Daily & Defense Report.
6. P. Feiler and D. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st ed. Reading, MA: Addison-Wesley, 2012.
7. M. W. Whalen, A. Gacek, D. D. Cofer, A. Murugesan, M. Per Erik Heimdahl, and S. Rayadurgam, "Your "what" is my "how": Iteration and hierarchy in system design," *IEEE Softw.*, vol. 30, no. 2, pp. 54–60, Mar.-Apr. 2013.
8. A. Gacek, et al., "Resolute: An assurance case language for architecture models," in *Proc. HILT 2014*, ACM, New York, pages 19–28.
9. A Cimatti, M Dorigatti, S Tonetta, "OCRA: A tool for checking the refinement of temporal contracts," in *Proc. ASE*, 2013, pp. 702–705.
10. A. Gacek, J. Backes, M. Whalen, L. G. Wagner, and E. Ghassabani, "The JKind Model Checker," *CAV*, no. 2, pp. 20–27, 2018.
11. *GSN Working Group*, GSN community standard version 1, 2011.
12. P. Graydon, J. Knight, and E. Strunk. "Assurance based development of critical systems," in *Proc. 2007 Int. Symp. Dependable Systems and Networks (DSN)*.
13. P. Hickey, et al. "Building embedded systems with embedded DSLs (experience report)," in *Proc. Int. Conf. Functional Programming (ICFP)*, ACM, 2014.
14. NASA Jet Propulsion Laboratory, "JPL institutional coding standard for the C programming language" Jet Propulsion Lab., Rep. JPL DOCID D-60411, 2009.
15. G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, pp. 2:1–2:70, Feb. 2014.
16. T. Nipkow, et al. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, vol. 2283 of LNCS. Heidelberg: Springer, 2002.
17. D. Cofer, A. Gacek, J. Backes, and K. Slind. "High-assurance cyber military systems (HACMS), 2017." Rockwell Collins. Accessed on: Aug., 15, 2018. [Online]. Available: https://insights .rockwellcollins.com/2017/07/06 /video-high-assurance-cyber-military-systems-hacms/