DOI:10.1145/3230627

Verified software secures the Unmanned Little Bird autonomous helicopter against mid-flight cyber attacks.

BY GERWIN KLEIN, JUNE ANDRONICK, MATTHEW FERNANDEZ, IHOR KUZ, TOBY MURRAY, AND GERNOT HEISER

Formally Verified Software in the Real World

IN FEBRUARY 2017, a helicopter took off from a Boeing facility in Mesa, AZ, on a routine mission around nearby hills. It flew its course fully autonomously, and the safety pilot, required by the Federal Aviation Administration, did not touch any controls during the flight. This was not the first autonomous flight of the AH-6, dubbed the Unmanned Little Bird (ULB);[3] it had been doing them for years. This time, however, the aircraft was subjected to mid-flight cyber attacks. The central mission computer was attacked by rogue camera software, as well as by a virus delivered through a compromised USB stick that had been inserted during maintenance. The attack compromised some subsystems but could not affect the safe operation of the aircraft.

One might think surviving such an attack is not a big deal, certainly that military aircraft would be robust against cyber attacks. In reality, a "red team" of professional penetration testers hired by the Defense Advanced Research Projects Agency (DARPA) under its High-Assurance Cyber Military Systems (HACMS) program had in 2013 compromised the baseline version of the ULB, designed for safety rather than security, to the point where it could have crashed it or diverted to any location of its choice. In this light, risking an in-flight attack with a human on board indicates that something had changed dramatically.

This article explains that change and the technology that enabled it. Specifically, it is about technology developed under the HACMS program, aiming to ensure the safe operation of critical real-world systems in a hostile cyber environment—multiple autonomous vehicles in this case. The technology is based on formally verified software, or software with machine-checked mathematical proofs it behaves according to its specification. While this article is not about the formal methods themselves, it explains how the verified artifacts can be used to secure practical systems. The most impressive outcome of HACMS is arguably that the technology could be retrofitted onto existing real-world systems, dramatically improving their cyber resilience, a process called "seismic security retrofit" in analogy to, say, the seismic retrofit of buildings. Moreover, most of the re-engineering

» key insights

- **Formal proof based on micro-kernel-enforced software architecture can scale to real systems at low cost.**
- **Mixed assurance levels and security levels within one system are possible and desirable; not all code has to be assured to the highest level.**
- **High assurance can be retrofitted to suitable existing systems with only moderate redesign and refactoring.**

Boeing Little Bird in unmanned flight test.

was done by Boeing engineers, not by formal verification researchers.

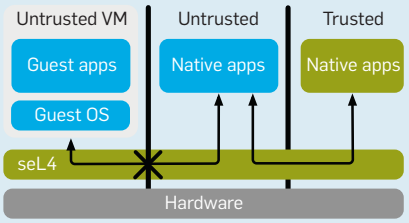
By far, not all the software on the HACMS vehicles was built on the basis of mathematical models and reasoning; the field of formal verification is not yet ready for such scale. However, HACMS demonstrated that significant improvement is feasible by applying formal techniques strategically to the most critical parts of the overall system. The HACMS approach works for systems in which the desired security property can be achieved through purely architecture-level enforcement. Its foundation is our verified microkernel, seL4, discussed later, which guarantees isolation between subsystems except for well-defined communication channels that are subject to the system's security policy. This isolation is leveraged by system-level component architectures that, through architecture, enforce the desired security property, and our verified component framework, CAmkES. The CAmkES framework integrates with architecture analysis tools from Rockwell Collins and the University of Minnesota, along with trusted high-assurance software components using domain-specific languages from Galois Inc.

The HACMS achievements are based on the software engineer's trusty old friend—modularization. What is new is that formal methods provide proof that interfaces are observed and module internals are encapsulated. This guaranteed enforcement of modularization allows engineers, like those at Boeing, who are not formal-method experts, to construct new or even retrofit existing systems, as discussed later, and achieve high resilience, even though the tools do not yet provide an overall proof of system security.

Formal Verification

Mathematical correctness proofs of programs go back to at least the 1960s,[14] but for a long time, their real-world benefit to software development was limited in scale and depth. However, a number of impressive breakthroughs have been seen in recent years in the formal code-level verification of real-life systems, from the verified C compiler CompCert[28] to the verified seL4 microkernel,[22,23,33] verified conference system CoCon,[21] verified ML compiler CakeML,[25] verified interactive theorem provers Milawa,[9] and Candle,[24] verified crash-resistant file system FSCQ,[5] verified distributed system IronFleet,[19] and verified concurrent kernel framework CertiKOS,[17] as well as significant mathematical theorems, including the Four Colour Theorem,[15] mechanized proof of the Kepler Conjecture,[18] and Odd Order Theorem.[16] None of these

Figure 1. Isolation and controlled communication with seL4.



are toy systems. For instance, CompCert is a commercial product, the seL4 microkernel is used in aerospace, autonomous aviation, and as an Internet of Things platform, and the CoCon system has been used in multiple full-scale scientific conferences.

These verification projects required significant effort, and for verification to be practical for widespread use, the effort needs to decrease. Here, we demonstrate how strategically combining formal and informal techniques, partially automating the formal ones, and carefully architecting the software to maximize the benefits of isolated components, allowed us to dramatically increase the assurance of systems whose overall size and complexity is orders-of-magnitude greater than that of the systems mentioned earlier.

Note we primarily use formal verification to provide proofs about correctness of code that a system's safety or security relies on. But it has other benefits as well. For example, code correctness proofs make assumptions about the context in which the code is run (such as behavior of hardware and configuration of software). Since formal verification makes these assumptions explicit, developer effort can focus on ensuring the assumptions hold—through other means of verification like testing. Moreover, in many cases systems consist of a combination of verified and non-verified code, and in them, formal verification acts as a lens, focusing review, testing, and debugging on the system's critical non-verified code.

seL4

We begin with the foundation for building provably trustworthy systems—the operating system (OS) kernel, the system's most critical part and enabler of cost-effective trustworthiness of the entire system.

The seL4 microkernel provides a formally verified minimal set of mechanisms for implementing secure systems. Unlike standard separation kernels[31] they are purposefully general and so can be combined for implementing a range of security policies for a range of system requirements.

One of the main design goals of seL4 (see the sidebar "Proof Effort") is to enforce strong isolation between mutually distrusting components that may run on top of it. The mechanisms support its use as a hypervisor to, say, host entire Linux operating systems while keeping them isolated from security-critical components that might run alongside, as outlined in Figure 1. In particular, this functionality allows system designers to deploy legacy components that may have latent vulnerabilities alongside highly trustworthy components.

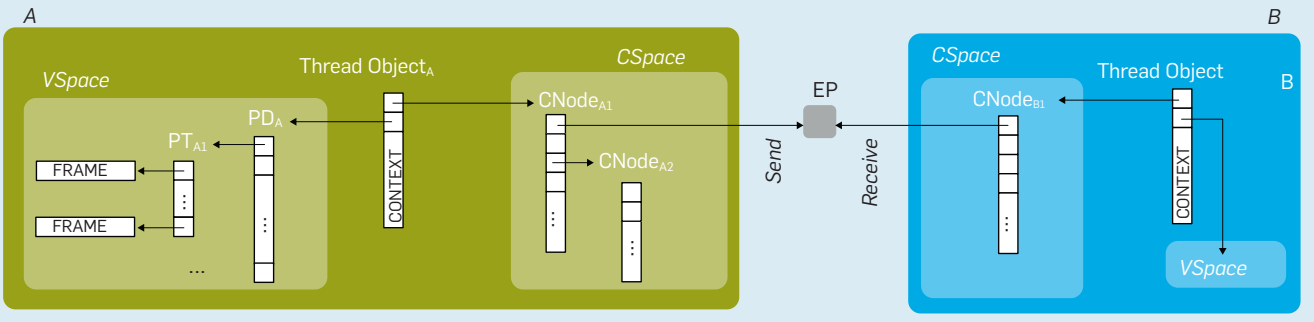
The seL4 kernel is unique among general-purpose microkernels. Not only does it deliver the best performance in its class,[20] its 10,000 lines of C code have been subjected to more formal verification than any other publicly available software in human history in terms not only of lines of proof but strength of properties proved. At the heart of this verification story sits the proof of "functional correctness" of the kernel's C implementation,[23] guaranteeing every behavior of the kernel is predicted by its formal abstract specification; see the online appendix (dl.acm.org/citation.cfm?doid=3230627&picked=formats) for an idea of how these proofs look. Following this guarantee, we added further proofs we explain after first introducing the main kernel mechanisms.

seL4 API. The seL4 kernel provides a minimal set of mechanisms for implementing secure systems: threads, capability management, virtual address spaces, inter-process communication (IPC), signaling, and interrupt delivery.

The kernel maintains its state in "kernel objects." For example, for each thread in a system there is a "thread object" that stores information about scheduling, execution, and access control. User-space programs can refer to kernel objects only indirectly through "capabilities"[10] that combine a reference to an object with a set of access rights to this object. For example, a thread cannot start or stop another thread unless it has a capability to the corresponding thread object.

Threads communicate and synchronize by sending messages through IPC "endpoint" objects. One thread with a send capability to an appropriate endpoint can message another thread that has a receive capability to that endpoint. "Notification" objects provide synchronization through sets of binary semaphores. Virtual address translation is managed by kernel objects that represent page directories,

Figure 2. Kernel objects for an example seL4-based system with two threads communicating via an endpoint.



page tables, and frame objects, or thin abstractions over the corresponding entities of the processor architecture. Each thread has a designated "VSpace" capability that points to the root of the thread's address-translation object tree. Capabilities themselves are managed by the kernel and stored in kernel objects called "CNodes" arranged in a graph structure that maps object references to access rights, analogous to page tables mapping virtual to physical addresses. Each thread has a distinguished capability identifying a root CNode. We call the set of capabilities reachable from this root the thread's "CSpace." Capabilities can be transmitted over endpoints with the grant operation and can be shared via shared CSpaces. Figure 2 outlines these kernel objects on an example.

Security proofs. With its generality, seL4's kernel API is necessarily low-level and admits highly dynamic system architectures. Direct reasoning about this API can thus be a challenge.

The higher-level concept of access control policies abstracts away from individual kernel objects and capabilities, capturing instead the access-control configuration of a system via a set of abstract "subjects" (think components) and the authorities each has over the others (such as to read data and send a message). In the example in Figure 2, the system would have components *A* and *B* with authority over the endpoint.

Sewell et al.[36] proved for such suitable access control policies that seL4 enforces two main security properties: authority confinement and integrity.

Authority confinement states that the access control policy is a static (unchanging) safe approximation of the concrete capabilities and kernel objects in the system for any future state of execution. This property implies that no matter how the system develops, no component will ever gain more authority than the access control policy predicts. In Figure 2, the policy for component *B* does not contain write access to component *A*, and *B* will thus never be able to gain this access in the future. The property thus implies that reasoning at the policy level is a safe approximation over reasoning about the concrete access-control state of the system.

Integrity states that no matter what a component does, it will never be able to modify data in the system (including by any system calls it might perform) the access control policy does not explicitly allow it to modify. For instance, in Figure 2, the only authority component *A* has over another component is the `send` right to the endpoint from which component *B* receives. This means the maximum state change *A* can effect in the system is in *A* itself and in *B*'s thread state and message buffer. It cannot modify any other parts of the system.

The dual of integrity is confidentiality, which states that a component cannot read another component's data without permission,[29] proved the stronger property of intransitive noninterference for seL4; that is, given a suitably configured system (with stronger restrictions than for integrity), no component is able to learn information about another component or its execution without explicit permission. The proof expresses this property in terms of an information-flow policy that can be extracted from the access-control policy used in the integrity proof. Information will flow only when explicitly allowed by the policy. The proof covers explicit information flows, as well as potential in-kernel covert storage channels, but timing channels are outside its scope and must be addressed through different means.[6]

Further proofs about seL4 include the extension of functional correctness, and thus the security theorems, to the binary level for the ARMv7 architecture[35] and a sound worst-case execution time profile for the kernel[2,34] necessary for real-time systems. The seL4 kernel is available for multiple architectures—ARMv6, ARMv7, ARMv7a, ARMv8, RISC-V, Intel x86, and Intel x64—and its machine-checked proof[33] is current on the ARMv7 architecture for the whole verification stack, as well as on ARMv7a with hypervisor extensions for functional correctness.

Proof Effort

seL4 design and code development took two person-years. Adding up all seL4-specific proofs over the years comes to a total of 18 person-years for 8,700 lines of C code. In comparison, L4Ka::Pistachio, another microkernel in the L4 family, comparable in size to seL4, took six person-years to develop and provides no significant level of assurance. This means there is only a factor 3.3 between verified software and traditionally engineered software. According to the estimation method by Colbert and Boehm,[8] a traditional Common Criteria EAL7 certification for 8,700 lines of C code would take more than 45.9 person-years. That means formal binary-level implementation verification is already more than a factor of 2.3 less costly than the highest certification level of Common Criteria yet provides significantly stronger assurance.

In comparison, the HACMS approach described here uses only these existing proofs for each new system, including the proofs generated from tools. The overall proof effort for a system that fits this approach is thus reduced to person-weeks instead of years, and testing can be significantly reduced to only validating proof assumptions.

Security by Architecture

The previous section summarized the seL4 kernel software engineers can use as a strong foundation for provably trustworthy systems. The kernel forms the bottom layer of the trusted computing base (TCB) of such systems. The TCB is the part of the software that needs to work correctly for the security property of interest to hold. Real systems have a much larger TCB than just the microkernel they run on, and more of the software stack would need to be formally verified to gain the same level of assurance as for the kernel. However, there are classes of systems for which this is not necessary, for which the kernel-level isolation theorems are already enough to enforce specific system-level security properties. This section includes an example of such a system.

The systems for which this works are those in which component architectures alone already enforce the critical property, potentially together with a few small, trusted components. Our example is the mission-control software of a quadcopter that was the research-demonstration vehicle in the HACMS program mentioned earlier.

Figure 3 outlines the quadcopter's main hardware components. It is intentionally more complex than needed for a quadcopter, as it is meant to be

representative of the ULB, and is, at this level of abstraction, the same as the ULB architecture.

The figure includes two main computers: a mission computer that communicates with the ground-control station and manages mission-payload software (such as for controlling a camera); and a flight computer with the task of flying the vehicle, reading sensor data, and controlling motors. The computers communicate via an internal network, a controller area network, or CAN bus, on the quadcopter, a dedicated Ethernet on the ULB. On the quadcopter, the mission computer also has an insecure WiFi link, giving us the opportunity to demonstrate further security techniques.

The subsystem under consideration in this example is the mission computer. Four main properties must be enforced: only correctly authenticated commands from the ground station are sent to the flight computer; cryptographic keys are not leaked; no additional messages are sent to the flight computer; and untrusted payload software cannot influence the vehicle's flight behavior. The operating assumption is that the camera is untrusted and potentially compromised, or malicious, that its drivers and the legacy payload software are potentially compromised, and any outside communication is likewise potentially compromised. For the purpose of this example, we assume a correct and strong cryptography implementation, or the key cannot be guessed, and that basic radio jamming and denial-of-service by overwhelming the ground station radio link are out of scope.

Figure 3. Autonomous-air-vehicle architecture.

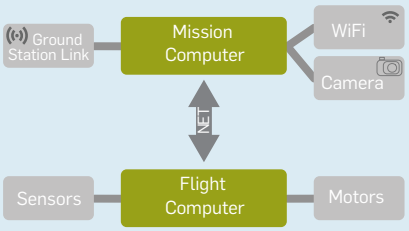
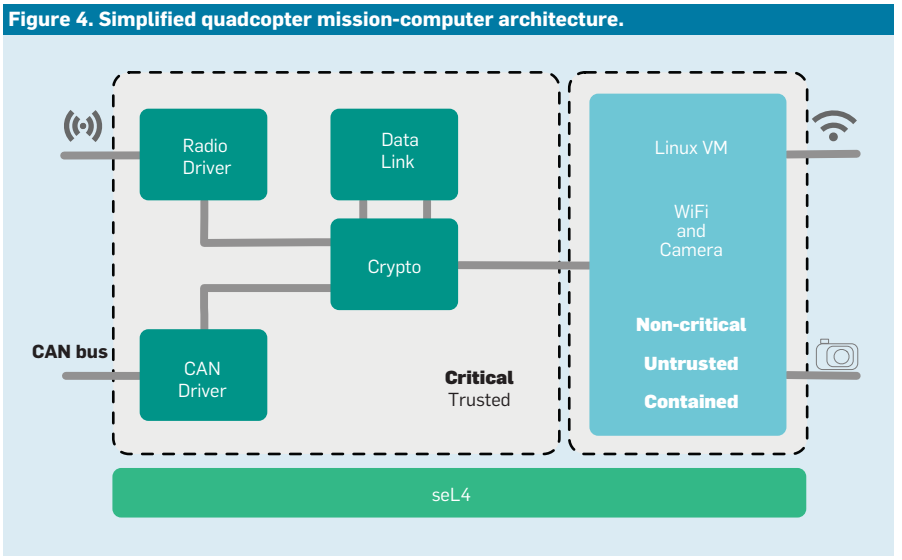


Figure 4 outlines how we design the quadcopter architecture to achieve these properties. We use a virtual machine (VM) running Linux as a containment vessel for legacy payload software, camera drivers, and WiFi link. We isolate the cryptography control module in its own component, with connections to the CAN bus component, to the ground station link, and to the Linux VM for sending image-recognition data back to the ground station. The purpose of the crypto component is to forward (only) authorized messages to the flight computer via the CAN interface stack and send back diagnostic data to the ground station. The radio-link component sends and receives raw messages that are encrypted, decrypted, and authenticated, respectively, by the crypto component.

Establishing the desired system properties is now reduced purely to the isolation properties and information-flow behavior of the architecture, and to the behavior of the single trusted crypto component. Assuming correct behavior of that component, keys cannot be leaked, as no other component has access to them; the link between Linux and the crypto component in Figure 4 is for message passing only and does not give access to memory. Only authenticated messages can reach the CAN bus, as the crypto component is the only connection to the driver. Untrusted payload software and WiFi are, as part of the Linux VM, encapsulated by component isolation and can communicate to the rest of the system only via the trusted crypto component.

Figure 4. Simplified quadcopter mission-computer architecture.



It is easy to imagine that this kind of architecture analysis could be automated to a high degree through model checking and higher-level mechanized reasoning tools. As observed in MILS systems,[1] component boundaries in an architecture are not just a convenient decomposition tool for modularity and code management but, with enforced isolation, provide effective boundaries for formal reasoning about the behavior of the system. However, the entire argument hinges on the fact that component boundaries in the architecture are correctly enforced at runtime in the final, binary implementation of the system.

The mechanisms of the seL4 kernel discussed earlier can achieve this enforcement, but the level of abstraction of the mechanisms is in stark contrast to the boxes and arrows of an architecture diagram; even the more abstract access-control policy still contains far more detail than the architecture diagram. A running system of this size contains tens of thousands of kernel objects and capabilities that are created programmatically, and errors in configuration could lead to security violations. We next discuss how we not only automate the configuration and construction of such code but also how we can automatically prove that architecture boundaries are enforced.

Verified Componentization

The same way reasoning about security becomes easier with the formal abstractions of security policies, abstraction also helps in building systems. The CAmkES component platform,[27] which runs on seL4 abstracts over the low-level kernel mechanisms, provides communication primitives, as well as support for decomposing a system into

functional units, as in Figure 5. Using this platform, systems architects can design and build seL4-based systems in terms of high-level components that communicate with each other and with hardware devices through connectors like remote procedure calls (RPCs), dataports, and events.

Generated code. Internally, CAmkES implements these abstractions using seL4's low-level kernel objects. Each component comprises (at least) one thread, a CSpace, and a VSpace. RPC connectors use endpoint objects, and CAmkES generates glue code to marshal and unmarshal messages and send them over IPC endpoints. Likewise, a dataport connector is implemented through shared memory, shared frame objects present in the address spaces of two components, and optionally restricting the direction of communication. Finally, an event connector is implemented using seL4's notification mechanism.

CAmkES also generates, in the capDL language,[26] a low-level specification of the system's initial configuration of kernel objects and capabilities. This capDL specification is the input for the generic seL4 initializer that runs as the first task after boot and performs the necessary seL4 operations to instantiate and initialize the system.[4]

In summary, a component platform provides free code. The component architecture describes a set of boxes and arrows, and the implementation task is reduced to simply filling in the boxes; the platform generates the rest while enforcing the architecture.

With a traditional component platform, the enforcement process would mean the generated code increases the trusted computing base of the system, as it has the ability to influence the functionality of the components. However, CAmkES also generates proofs.

Automated proofs. While generating glue code, CAmkES produces formal proofs in Isabelle/HOL, following a translation-validation approach,[30] demonstrating that the generated glue code obeys a high-level specification and the generated capDL specification is a correct refinement of the CAmkES description.[12] We have also proved that the generic seL4 initializer correctly sets up the system in the desired initial configuration. In doing so, we automate large parts of system construction without expanding the trusted computing base.

Figure 5. CAmkES workflow.

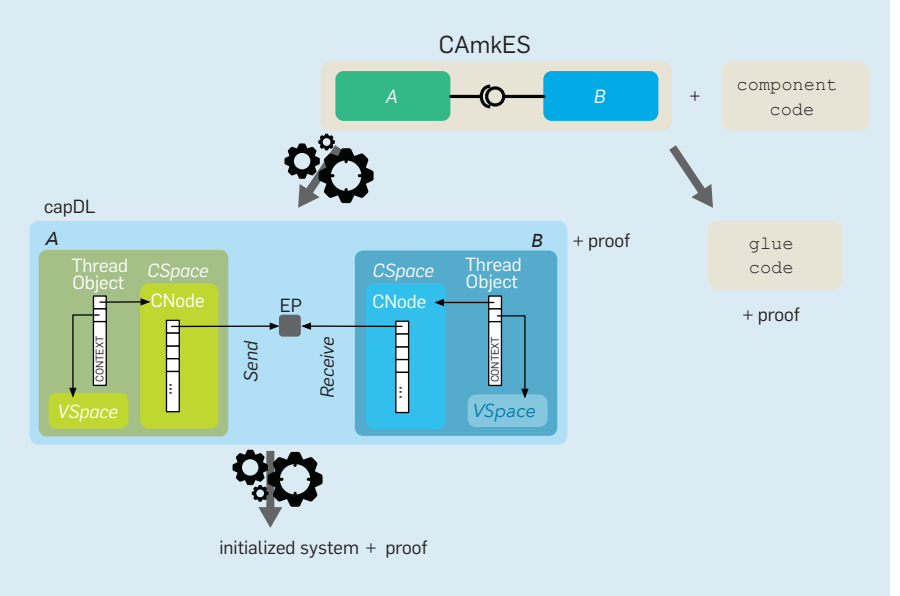
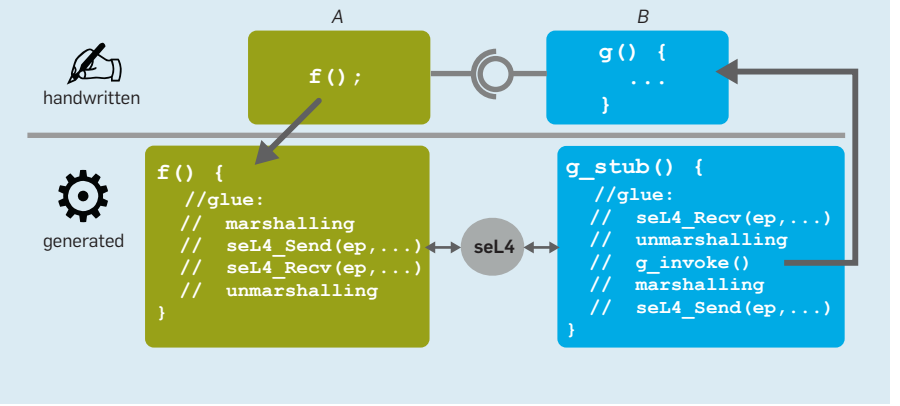


Figure 6. RPC-generated code.



Developers rarely look at the output of code generators, focusing instead on the functionality and business logic of their systems. In the same way, we intend the glue code proofs to be artifacts that do not need to be examined, meaning developers can focus on proving the correctness of their handwritten code. Mirroring the way a header generated by CAmkES gives the developer an API for the generated code, the top-level generated lemma statements produce a proof API. The lemmas describe the expected behavior of the connectors. In the example of RPC glue code outlined in Figure 6, the generated function *f* provides a way to invoke a remote function *g* in another component. To preserve the abstraction, calling *f* must be equivalent to calling *g*. The lemma the system generates ensures the invocation of the generated RPC glue code *f* behaves as a direct invocation of *g*, as if it were co-located with the caller.

To be useful, the proofs the system generates must be composable with (almost) arbitrary user-provided proofs, both of the function *g* and of the contexts where *g* and *f* are used. To enable this composability, the specification of the connectors is parameterized through user-provided specifications of remote functions. In this way, proof engineers can reason about their architecture, providing specifications and proofs for their components, and rely on specifications for the generated code.

To date, we have demonstrated this process end-to-end using a specific CAmkES RPC connector.[12,13] Extending the proof generator to support other

connectors, allowing construction of more diverse verified systems, should be simpler to achieve, because other connector patterns (data ports and events) are significantly less complex than RPC.

Next to communication code, CAmkES produces the initial access control configuration that is designed to enforce architecture boundaries. To prove the two system descriptions—capDL and CAmkES—correspond, we consider the CAmkES description as an abstraction of the capDL description. We use the established framework[36] mentioned earlier to infer authority of one object over another object from a capDL description to lift reasoning to a policy level. Additionally, we have defined rules for inferring authority between components in a CAmkES description. The produced proof ensures the capDL objects, when represented as an authority graph with objects grouped per component, have the same intergroup edges as the equivalent graph between CAmkES components.[12] Intuitively, this correspondence between the edges means an architecture analysis of the policy inferred by the CAmkES description will hold for the policy inferred by the generated capDL description, which in turn is proved to satisfy authority confinement, integrity, and confidentiality, as mentioned earlier.

Finally, to prove correct initialization, CAmkES leverages the generic initializer that will run as the first user task following boot time. In seL4, this first (and unique) user task has access to all available memory, using it to create objects and capabilities according to the detailed capDL description it takes as input. We proved that the state following execution of the initializer satisfies the one described in the given specification.[4] This proof holds for a precise model of the initializer but not yet at the implementation level. Compared to the depth of the rest of the proof chain, this limitation may appear weak, but it is already more formal proof than would be required for the highest level (EAL7) of a Common Criteria security evaluation.

Seismic Security Retrofit

In practice, there are few opportunities to engineer a system from scratch for security, so the ability to retrofit for security is crucial for engineering secure systems. Our seL4-based framework supports an iterative process we call "seismic security retrofit," as a regular structural architect might retrofit an existing building for greater resilience against earthquakes. We illustrate the process by walking through an example that incrementally adapts the existing software architecture of an autonomous air vehicle, moving it from a traditional testing approach to a high-assurance system with theorems backed by formal methods. While this example is based on work done for a real vehicle—the ULB—it is simplified for presentation and does not include all details.

The original vehicle architecture is the same as the architecture outlined in Figure 3. Its functionality is split over two separate computers: a flight computer that controls the actual flying and the mission computer that performs high-level tasks (such as ground-station communication and camera-based navigation). The original version of the mission computer was a monolithic software application running on Linux. The rest of the example concentrates on a retrofit of this mission-computer functionality. The system was built and re-engineered by Boeing engineers, using the methods, tools, and components provided by the HACMS partners.

Step 1. Virtualization. The first step was to take the system as is and run it in a VM on top of a secure hypervisor (see Figure 7). In the seismic-retrofit metaphor, doing so corresponds to situating the system on a more flexible foundation. A VM on top of seL4 in this system consists of one CAmkES component that includes a virtual machine monitor (VMM) and the guest operating system, in this case Linux. The kernel provides abstractions of the virtualization hardware, while the VMM manages these abstractions for the VM. The seL4 kernel constrains not only the guest but also the VMM, so the VMM implementation does not need to be trusted to enforce isolation. Failure of the VMM will lead to failure of the guest but not to failure of the complete system.

Depending on system configuration, the VM may have access to hardware devices through para-virtualized drivers, pass-through drivers, or both. In the case of pass-through drivers, developers can make use of a system MMU or IOMMU to prevent hardware devices and drivers in the guest from breaching isolation boundaries. Note that simply running a system in a VM adds no additional security or reliability benefits. Instead, the reason for this first step is to enable step 2.

Step 2. Multiple VMs. The second step in a seismic retrofit strengthens existing walls. In software, the developer can improve security and reliability by splitting the original system into multiple subsystem partitions, each consisting of a VM running the

Figure 7. All functionality in a single VM.

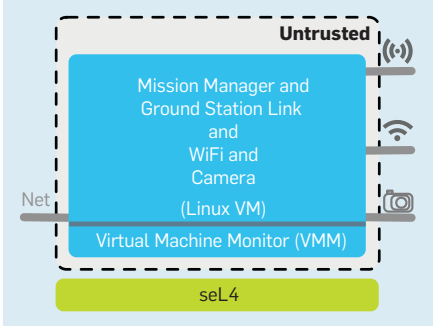


Figure 8. Functionality split into multiple VMs.

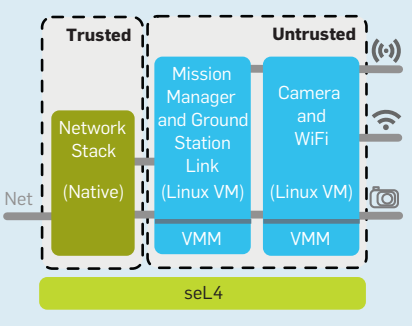
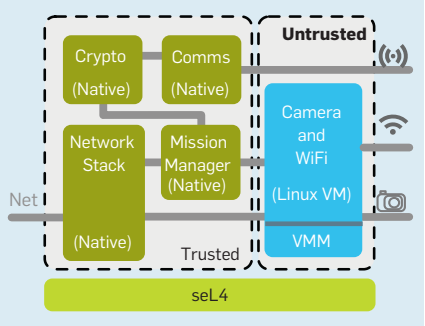


Figure 9. Functionality in native components.



code of only part of the original system. Each VM/VMM combination runs in a separate CAmkES component that introduces isolation between the different subsystems, keeping mutually distrusting ones from affecting each other, and, later, allowing different assurance levels to coexist.

In general, the partitions follow the existing software architecture, although a redesign may be necessary where the software architecture is inadequate for effective isolation.

The partitions will in general need to communicate with each other, so in this step we also add appropriate communication channels between them. For security, it is critically important that these interfaces are narrow, limiting the communication between partitions to only what is absolutely necessary to maximize the benefits of isolation. Moreover, interface protocols should be efficient, keeping the required number of messages or amount of data copying minimal. Critically, seL4's ability to enable controlled and limited sharing of memory between partitions allows a developer to minimize the amount of data copying.

Besides the VMs that represent subsystems of the original system, we also extract and implement components for any shared resources (such as the network interface).

We can iterate the entire step 2 until we have achieved the desired granularity of partitions. The right granularity is a trade-off between the strength of isolation on the one hand and the increased overhead and cost of communication between partitions, as well as re-engineering cost, on the other.

In our example we end up with three partitions: a VM that implements the ground-station communication functionality running on Linux; another VM that implements camera-based navigation functionality (also running on Linux); and a native component that provides shared access to the network, as in Figure 8.

Step 3. Native components. Once the system has been decomposed into separate VM partitions, some or all of the individual partitions can be reimplemented as native components rather than as VMs. The aim is to significantly reduce the attack surface for the same functionality. An additional benefit of transforming a component into native code is a reduced footprint and better performance, removing the guest operating system and removing the execution and communication overhead of the VMM.

We intend the glue code proofs to be artifacts that do not need to be examined, meaning developers can focus on proving the correctness of their handwritten code.

Using a native component also increases the potential for applying formal verification and other techniques for improving the assurance and trustworthiness of the component. Examples range from full functional verification of handwritten code to cogeneration of code and proofs, application of model checking, using type-safe programming languages, and static analysis or traditional thorough testing of a smaller codebase.

Due to the isolation provided by seL4 and the componentized architecture, it becomes possible for components of mixed assurance levels to coexist in the system without decreasing the overall assurance to that of the lowest-assurance component or increasing the verification burden of the lowest-assurance components to that of the highest-assurance ones.

In our example, we target the VM for mission manager and ground-station link, implementing the communications, cryptography, and mission-manager functionality as native components. We leave the camera and WiFi to run in a VM as an untrusted legacy component (see Figure 9). This split was a trade-off between the effort to reimplement the subsystems and the benefit gained by making them native from both a performance and an assurance perspective.

Step 4. Overall assurance. With all parts in place, the final step is to analyze the assurance of the overall system based on the assurance provided by the architecture and by individual components.

In HACMS, the communication, cryptography, and mission manager functionality were implemented in a provably type-safe, domain-specific language called Ivory,[11] with fixed heap-memory allocation. Without further verification, Ivory does not give us high assurance of functional correctness but does give us assurance about robustness and crash-safety. Given component isolation, we reason that these assurances are preserved in the presence of untrusted components (such as the camera VM).

The networking component is implemented in standard C code consisting of custom code for the platform and pre-existing library code. Its assurance level corresponds to that obtained through careful implementation of known code. Robustness could be increased without much cost through such techniques as driver synthesis[32] and type-safe languages, as with Ivory. However, in the overall security analysis of the system, any compromise of the network component would be able to inject or modify only network packets. Since the traffic is encrypted, such an attack would not compromise the guarantee that only authorized commands reach the flight computer.

The camera VM is the weakest part of the system, since it runs a stock Linux system and is expected to have vulnerabilities. However, as the VM is isolated, if attackers were to compromise the VM, they would not be able to escape to other components. The worst an attacker could do is send incorrect data to the mission-manager component. As in the quadcopter, the mission manager validates data it receives from the camera VM. This is the part of the system on the ULB that demonstrated containment of a compromise in the in-flight attack mentioned at the beginning of the article. This was a white-box attack, where the Red Team had access to all code and documentation, as well as to all external communication, and was intentionally given root access to the camera VM, simulating a successful attack against legacy software. Successfully containing the attack and being able to defend against this very powerful Read Team scenario served to validate the strength of our security claims and uncover any missed assumptions, interface issues, or other security issues the research team might have failed to recognize.

The camera VM is the weakest part of the system, since it runs a stock Linux system and is expected to have vulnerabilities.

Limitations and Future Work

This article has given an overview of a method for achieving very high levels of assurance for systems in which security property can be enforced through their component architecture. We have proved theorems for the kernel level and its correct configuration, as well as theorems that ensure the component platform correctly configures protection boundaries according to its architecture description, and that it produces correct RPC communication code. The connection with a high-level security analysis of the system remains informal, and the communication code theorems do not cover all communication primitives the platform provides. While more work would be required to automatically arrive at an end-to-end system-level theorem, it is clear at this stage that one is feasible.

The main aim of the reported work is to dramatically reduce verification effort for specific system classes. While the purely architecture-based approach described here can be driven a good deal further than in the ULB example, it is clearly limited by the fact it can express only properties that are enforced by the component architecture of the system. If that architecture changes at runtime or if the properties of interest critically depend on the behavior of too many or too-large trusted components, returns will diminish.

The first step to loosen these limitations would be a library of pre-verified high-assurance components for use as trusted building blocks in such architectures. This library could include security patterns (such as input sanitizers, output filters, down-graders, and runtime monitors) potentially generated from higher-level specifications but also such infrastructure components as reusable crypto modules, key storage, file systems, network stacks, and high-assurance drivers. If the security property depends on more than one such component, it would become necessary to reason about the trustworthiness of their interaction and composition. The main technical challenges here are concurrency reasoning, protocols, and information-flow reasoning in the presence of trusted components. Despite these limitations, this work demonstrates that the rapid development of real high-assurance seL4-based systems is now a reality that can be achieved for a cost that is lower than traditional testing.

Acknowledgments

We are grateful to Kathleen Fisher, John Launchbury, and Raymond Richards for their support as program managers in HACMS, in particular Kathleen

Fisher for having the vision to start the program. John Launchbury coined the term "seismic security retrofit." We thank Lee Pike for feedback on an earlier draft. We would also like to acknowledge our HACMS project partners from Rockwell Collins, the University of Minnesota, Galois, and Boeing. While we concentrated on the operating system aspects of the HACMS project here, the rapid construction of high-assurance systems includes many further components, including a trusted build, as well as architecture and security-analysis tools. This material is based on research sponsored by the U.S. Air Force Research Laboratory and the Defense Advanced Research Projects Agency under agreement number FA8750-12-9-0179. The U.S. government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory, Defense Advanced Research Projects Agency, or U.S. government. C

References

1. Alves-Foss, J., Oman, P.W., Taylor, C., and Harrison, S. The MILS architecture for high-assurance embedded systems. *International Journal of Embedded Systems 2*, 3-4 (2006), 239–247.
2. Blackham, B., Shi, Y., Chattopadhyay, S., Roychoudhury, A., and Heiser, G. Timing analysis of a protected operating system kernel. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium* (Vienna, Austria, Nov. 29–Dec. 2). IEEE Computer Society Press, 2011, 339–348.
3. Boeing. Unmanned Little Bird H-6U; http://www.boeing.com/defense/unmanned-little-bird-h-6u/
4. Boyton, A., Andronick, J., Bannister, C., Fernandez, M., Gao, X., Greenaway, D., Klein, G., Lewis, C., and Sewell, T. Formally verified system initialisation. In *Proceedings of the 15th International Conference on Formal Engineering Methods* (Queenstown, New Zealand, Oct. 29–Nov. 1). Springer, Heidelberg, Germany, 2013 70–85.
5. Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Frans Kaashoek, M., and Zeldovich, N. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles* (Monterey, CA, Oct. 5–7). ACM Press, New York, 2015, 18–37.
6. Cock, D., Ge, Q., Murray, T., and Heiser, G. The last mile: An empirical study of some timing channels on seL4. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, AZ, Nov. 3–7). ACM Press, New York, 2014, 570–581.
7. Cock, D., Klein, G., and Sewell, T. Secure microkernels, state monads and scalable refinement. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics* (Montreal, Canada, Aug. 18–21). Springer, Heidelberg, Germany, 2008, 167–182.
8. Colbert, E. and Boehm, B. Cost estimation for secure software & systems. In *Proceedings of the International Society of Parametric Analysts / Society of Cost Estimating and Analysis 2008 Joint International Conference* (Noordwijk, the Netherlands, May 12–14). Curran, Red Hook, NY, 2008.
9. Davis, J. and Myreen, M.O. The reflective Milawa theorem prover is sound (down to the machine code that runs it). *Journal of Automated Reasoning 55*, 2 (Aug. 2015), 117–183.
10. Dennis, J.B. and Van Horn, E.C. Programming semantics for multi-programmed computations. *Commun. ACM 9*, 3 (Mar. 1966), 143–155.
11. Elliott, T., Pike, L., Winwood, S., Hickey, P., Bielman, J., Sharp, J., Seidel, E., and Launchbury, J. Guilt-free Ivory. In *Proceedings of the ACM SIGPLAN Haskell Symposium* (Vancouver, Canada, Sept. 3–4). ACM Press, New York, 189–200.
12. Fernandez, M. *Formal Verification of a Component Platform.* Ph.D. thesis. School of Computer Science & Engineering, University of New South Wales, Sydney, Australia, July 2016.
13. Fernandez, M., Andronick, J., Klein, G., and Kuz, I. Automated verification of RPC stub code. In *Proceedings of the 20th International Symposium on Formal Methods* (Oslo, Norway, June 22–26). Springer, Heidelberg, Germany, 2015, 273–290.
14. Floyd, R.W. Assigning meanings to programs. *Mathematical Aspects of Computer Science 19*, (1967), 19–32.
15. Gonthier, G. *A Computer-Checked Proof of the Four-Colour Theorem.* Microsoft Research, Cambridge, U.K, 2005; https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/gonthier-4colproof.pdf
16. Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Le Roux, S., Mahboubi, A., O'Connor, R., Biha S.O., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., and Théry, L. A machine-checked proof of the Odd Order Theorem. In *Proceedings of the Fourth International Conference on Interactive Theorem Proving, Volume 7998 of LNCS* (Rennes, France, July 22–26). Springer, Heidelberg, Germany, 2013, 163–179.
17. Gu, R., Shao, Z., Chen, H., Wu, X.(N.)., Kim, J., Sjöberg, V., and Costanzo, C. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (Savannah, GA, Nov. 2–4). ACM Press, New York, 2016.
18. Hales, T.C., Adams, M., Bauer, G., Dang, D.T., Harrison, J., Le Hoang, T., Kaliszyk, C., Magron, V., McLaughlin, S., Nguyen, T.T., Nguyen, T.Q., Nipkow, T., Obua, S., Pleso, J., Rute, J., Solovyev, A., Ta, A.H.T., Tran, T.N., Trieu, T.T., Urban, J., Vu, K.K., and Zumkeller, R. A formal proof of the Kepler Conjecture. *Forum of Mathematics, Pi, Volume 5.* Cambridge University Press, 2017.
19. Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J.R., Parno, B., Roberts, M.L., Setty, S.T.V., and Zill, B. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles* (Monterey, CA, Oct. 5–7). ACM Press, New York, 2015, 1–17.
20. Heiser, G. and Elphinstone, K. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Transactions on Computer Systems 34*, 1 (Apr. 2016), 1:1–1:29.
21. Kanav, S., Lammich, P., and Popescu, A. A conference management system with verified document confidentiality. In *Proceedings of the 26th International Conference on Computer Aided Verification* (Vienna, Austria, July 18–22). ACM Press, New York, 2014, 167–183.
22. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., and Heiser, G. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems 32*, 1 (Feb. 2014), 2:1–2:70.
23. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (Big Sky, MT, Oct. 11–14). ACM Press, New York, 2009, 207–220.
24. Kumar, R., Arthan, R., Myreen, M.O., and Owens, S. Self-formalisation of higher-order logic: Semantics, soundness, and a verified implementation. *Journal of Automated Reasoning 56*, 3 (Apr. 2016), 221–259.
25. Kumar, R., Myreen, M., Norrish, M., and Owens, S. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, CA, Jan. 22–24). ACM Press, New York, 2014, 179–191.
26. Kuz, I., Klein, G., Lewis, C., and Walker, A. capDL: A language for describing capability-based systems. In *Proceedings of the First ACM Asia-Pacific Workshop on Systems* (New Delhi, India, Aug. 30–Sept. 3). ACM Press, New York, 2010, 31–35.
27. Kuz, I., Liu, Y., Gorton, I., and Heiser, G. CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software (Special Edition on Component-Based Software Engineering of Trustworthy Embedded Systems) 80*, 5 (May 2007), 687–699.
28. Leroy, X. Formal verification of a realistic compiler. *Commun. ACM 52*, 7 (July 2009), 107–115.
29. Murray, T., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X., and Klein, G. seL4: From general-purpose to a proof of information flow enforcement. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (San Francisco, CA, May 19–22). IEEE Press, Los Alamitos, CA, 2013, 415–429.
30. Pnueli, A., Siegel, M., and Singerman, E. Translation validation. In *Proceedings of the Fourth International Conference on Tools and Algorithms for Construction and Analysis of Systems* (Lisbon, Portugal, Mar. 28–Apr. 4). Springer, Berlin, Germany, 1998, 151–166.
31. Rushby, J. Design and verification of secure systems. In *Proceedings of the Eighth Symposium on Operating System Principles* (Pacific Grove, CA, Dec. 14–16). ACM Press, New York, 1981, 12–21.
32. Ryzhyk, L., Chubb, P., Kuz, I., Le Sueur, E., and Heiser, G. Automatic device driver synthesis with Termite. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (Big Sky, MT, Oct. 11–14). ACM Press, New York, 2009, 73–86.
33. seL4 microkernel code and proofs; https://github.com/seL4/
34. Sewell, T., Kam, F., and Heiser, G. Complete, high-assurance determination of loop bounds and infeasible paths for WCET analysis. In *Proceedings of the 22nd IEEE Real Time and Embedded Technology and Applications Symposium* (Vienna, Austria, Apr. 11–14). IEEE Press, 2016.
35. Sewell, T., Myreen, M., and Klein, G. Translation validation for a verified OS kernel. In *Proceedings of the 34th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, WA, June 16–22). ACM Press, New York, 2013, 471–481.
36. Sewell, T., Winwood, S., Gammie, P., Murray, T., Andronick, J., and Klein, G. seL4 enforces integrity. In *Proceedings of the International Conference on Interactive Theorem Proving* (Nijmegen, the Netherlands, Aug. 22–25). Springer, Heidelberg, Germany, 2011, 325–340.

Gerwin Klein (gerwin.klein@data61.csiro.au) is a Chief Research Scientist at Data61, CSIRO, and Conjoint Professor at UNSW, Sydney, Australia.

June Andronick (june.andronick@data61.csiro.au) is a Principal Research Scientist at Data61, CSIRO, Conjoint Associate Professor at UNSW, Sydney, Australia, and the leader of the Trustworthy Systems group at Data61, known for the formal verification of the seL4 operating system microkernel.

Matthew Fernandez (matthew.fernandez@gmail.com) participated in this project while he was a Ph.D. student at UNSW, Sydney, Australia, and is today a researcher at Intel Labs, USA.

Ihor Kuz (ihor.kuz@data61.csiro.au) is a Principal Research Engineer at Data61, CSIRO , and also a Conjoint Associate Professor at UNSW, Sydney, Australia.

Toby Murray (toby.murray@unimelb.edu.au) is a lecturer at the University of Melbourne, Australia, and a Senior Research Scientist at Data61, CSIRO.

Gernot Heiser (gernot@unsw.edu.au) is a Scientia Professor and John Lions Chair of Computer Science at UNSW, Sydney, Australia, a Chief Research Scientist at Data61, CSIRO, and a fellow of the ACM, the IEEE, and the Australian Academy of Technology and Engineering.