Formally Verified Software in the Real World

Gerwin Klein
Data61, CSIRO
and
UNSW
Sydney, Australia

June Andronick
Data61, CSIRO
and
UNSW
Sydney, Australia

Matthew Fernandez
Data61, CSIRO
and
UNSW[*]
Sydney, Australia

Ihor Kuz
Data61, CSIRO
and
UNSW
Sydney, Australia

Toby Murray
Data61, CSIRO
and
University of Melbourne
Melbourne, Australia

Gernot Heiser
Data61, CSIRO
and
UNSW
Sydney, Australia

ABSTRACT

We present an approach for building highly-dependable systems that derive their assurance from a formally-verified operating-system which guarantees isolation between subsystems. We leverage those guarantees to enforce security through non-bypassable architectural constraints, and through generation of code and proofs from the architecture. We show that this approach can produce a system that is highly robust against cyber attacks, even without formal proof of its overall security. We demonstrate not only that this approach is applicable to real-world systems, such as autonomous vehicles, but also that it is possible to re-engineer an existing insecure system to achieve high robustness, and that this can be done by engineers not trained in formal methods.

CCS Concepts

•**Software and its engineering** → **Operating systems; Software verification;** •**Security and privacy** → *Formal methods and theory of security;*

Keywords

Provably trustworthy systems, formal verification, seL4

1. INTRODUCTION

In February 2017, a helicopter took off from a Boeing facility in Mesa, AZ, to fly a routine mission around nearby hills. The helicopter flew its course fully autonomously, the safety pilot, required by the FAA, 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) [Boeing], it had been doing those 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 a virus implanted from a compromised USB stick during maintenance. The attack compromised some subsystems, but could not affect the safe operation of the aircraft.

One might think that this is not a big deal, certainly military aircraft would be robust against cyber attacks? Yet, reality is different: In 2013, a "Red Team" of professional penetration testers, hired by DARPA under the *High-Assurance Cyber Military Systems* (HACMS) program, compromised the baseline version of the ULB, designed for safety rather than security, to the point where they could have crashed it or diverted to any location of their choice. In that 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, aimed at ensuring the safe operation of critical real-world systems in a hostile cyber environment, various autonomous vehicles in our case. The technology is based on formally verified software, software with machine-checked mathematical proofs that it behaves according to its specification. While this article is not about the formal methods themselves, it explains how the verified artefacts can be used to secure practical systems. Arguably the most impressive outcome of HACMS is that this technology could be retrofitted to existing real-world systems, dramatically improving their cyber-resilience, a process dubbed *seismic security retrofit* in analogy to the seismic retrofit of buildings. Importantly, most of the re-engineering of the system 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 is not yet ready for that. However, HACMS demonstrated that significant improvements are possible, and feasible, by applying such techniques strategically, to the most critical parts of the overall system.

The HACMS approach works for systems where the desired security property can be achieved purely by architecture-level enforcement. Its foundation is our verified microkernel, seL4 (Section 3), which guarantees isolation between subsystems, except for well-defined communication channels that are subject to the system's security policy. This is supported by system-level component architectures that, by architecture, enforce the desired security property (Section 4), and our verified component framework, CAmkES (Section 5). The CAmkES framework integrates with architecture analysis tools from Rockwell Collins and the University of Minnesota, and trusted high-assurance software components using domain-specific languages from Galois.

[*] Now at Intel Corporation.

ACM ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235

The HACMS achievements are based on the software engineer's old friend, modularisation. What is new is that formal methods provide *proof* that interfaces are observed and module internals are encapsulated. This allows engineers, such as Boeing's, who are not formal-method experts, to construct new or even retrofit existing systems (Section 6), and achieve high resiliency, even though the tools do not yet provide an overall proof of system security.

2. FORMAL VERIFICATION

Mathematical correctness proofs of programs go back to at least the 1960s [Floyd, 1967], but for a long time, real-world impact was limited in scale and depth. However, recent years have seen a number of impressive breakthroughs in the formal code-level verification of real-life systems. These range from the verified C compiler CompCert [Leroy, 2009], the verified seL4 microkernel [Klein et al., 2009, 2014; seL4], the verified conference system CoCon [Kanav et al., 2014], the verified ML compiler CakeML [Kumar et al., 2014], the verified interactive theorem provers Milawa [Davis and Myreen, 2015] and Candle [Kumar et al., 2016], the verified crash resistant file system FSCQ [Chen et al., 2015], the verified distributed system IronFleet [Hawblitzel et al., 2015], to the verified concurrent kernel framework CertiKOS [Gu et al., 2016], and more — not to mention significant mathematical theorems such as the Four Colour Theorem [Gonthier, 2005], the mechanised proof of the Kepler Conjecture [Hales et al., 2015], and the Odd Order Theorem [Gonthier et al., 2013]. These are not toy systems. For instance, CompCert is a commercial product, the seL4 microkernel is used amongst others, in aerospace, autonomous aviation, and as an internet of things platform; the CoCon system has been used in practice for multiple full-scale scientific conferences.

All of these verifications required significant effort, and for verification to become practical for wide-spread use, this effort needs to decrease. In this article we demonstrate how strategically combining formal with informal techniques, partially automating the formal ones, and carefully architecting the software to maximise the benefits of isolated components, allows us to dramatically increase the assurance of systems whose overall size and complexity is orders of magnitude larger than that of the systems listed above.

Note that we primarily use formal verification to provide proofs about correctness of code that we rely upon, however, it has other benefits as well. Code correctness proofs have assumptions about the context in which the code is run (e.g. behaviour of hardware, configuration of software, etc.). Since these assumptions are made explicit, effort can be targeted at ensuring that the assumptions hold (through other means of verification such as testing). Furthermore, in many cases systems will consist of a combination of verified and non-verified code. In such cases formal verification acts as a lens, focussing review, testing, and debugging attention on critical non-verified code in the system.

3. SEL4

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

The seL4 microkernel provides a formally-verified, minimal set of *mechanisms* for implementing secure systems. Compared to standard separation kernels [Rushby, 1981], these mechanisms are purposefully general, and can be combined to implement a wide range of security *policies* for a wide range of system requirements.

One of seL4's main design goals is to enforce strong isolation between mutually distrusting components that may run on top of it. The mechanisms support its use as a hypervisor, for instance, to host entire Linux operating systems while keeping them isolated from security-critical components that might run alongside, as depicted in Figure 1. In particular this functionality allows us to deploy legacy components, which may have latent vulnerabilities, alongside highly trustworthy components.

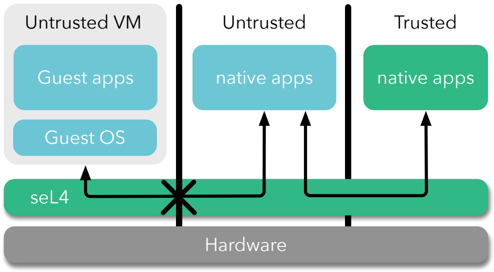


Figure 1: Isolation and controlled communication with seL4

The seL4 kernel is unique amongst general-purpose microkernels. Not only does it have the highest performance in its class [Heiser and Elphinstone, 2016], but its 10,000 lines of C code also have been subject to more formal verification than any other publicly available piece of software in human history (not only measured by lines of proof). At the heart of seL4's proofs sits the proof of *functional correctness* of the kernel's C implementation [Klein et al., 2009]. This proof guarantees that every behaviour of the kernel is predicted by its formal *abstract specification*. Following this, we added further proofs, which we explain below after introducing the main kernel mechanisms.

3.1 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), signalling, 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 only refer to kernel objects indirectly through *capabilities* [Dennis and Van Horn, 1966]. A capability combines 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 possesses a capability to the corresponding thread object.

Threads communicate and synchronise by sending messages through IPC *endpoint* objects. One thread with a Send capability to an appropriate endpoint can message another thread which has a Receive capability to that endpoint. *Notification* objects provide asynchronous communication through sets of binary semaphores.

Virtual address translation is managed by kernel objects that represent page directories, page tables, and frame objects. These are thin abstractions over the corresponding entities of the processor architecture. Each thread possesses 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, similarly to page tables mapping virtual to physical addresses. Each thread possesses a distinguished capability identifying a root CNode. We call the set of capabilities reachable from the root the thread's *CSpace*. Capabilities can be transmitted over endpoints with the *grant* operation, and they can be shared via shared CSpaces.

Figure 2 illustrates these kernel objects on an example.

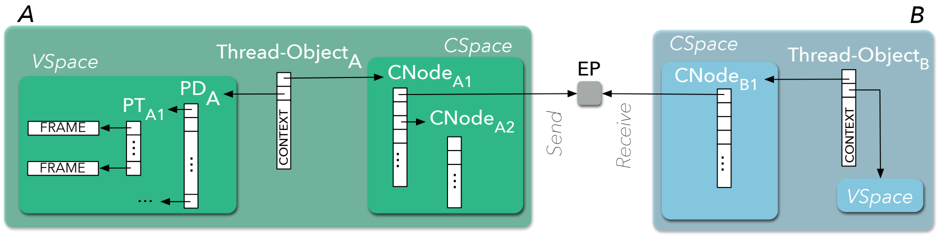


Figure 2: Kernel objects for an example seL4-based system with two threads communicating via an endpoint

3.2 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 therefore be relatively involved.

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

Sewell et al. [2011] proved for suitable such 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 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 therefore B will never be able to gain this access in the future. This means 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) that 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 component B receives from. 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. Murray et al. [2013] proved the stronger property of intransitive non-interference for seL4: given a suitably configured system (with stronger restrictions than for integrity), no component will be able to learn information about another component or its execution without explicit permission. The proof expresses this in terms of an information-flow policy that can be extracted from the access control policy used in the integrity proof. Information will only flow when explicitly allowed by this policy. This proof covers explicit information flows as well as potential in-kernel covert storage channels, but timing-channels are outside of its scope and must be addressed using different means [Cock et al., 2014].

Further proofs about seL4 include the extension of functional correctness, and thereby the security theorems to the binary level for the ARMv7 architecture [Sewell et al., 2013], and a sound worst-case execution time profile for the kernel [Blackham et al., 2011; Sewell et al., 2016], necessary for real-time systems.

The seL4 kernel is available for multiple architectures (ARMv6, ARMv7, ARMv7a, ARMv8, Intel x86, and Intel x64), and its machine-checked proof [seL4] is current on the ARMv7 architecture for the whole verification stack, as well as on ARMv7a with hypervisor extensions for functional correctness.

4. SECURITY BY ARCHITECTURE

The previous section summarised the seL4 kernel, which we 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. Of course, 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 shows an example of such a system.

The systems for which this works are those whose 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, which was the research demonstration vehicle in the aforementioned HACMS program.

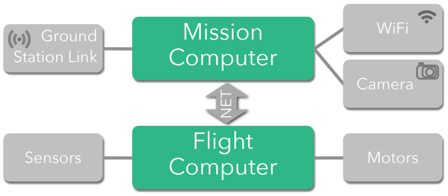


Figure 3: Autonomous air vehicle architecture

Figure 3 shows the main hardware components of this quadcopter. It is intentionally more complex than needed for a quadcopter, because it is meant to be representative of the ULB. At this level of abstraction, it is the same as the ULB architecture.

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

The sub-system under consideration in this example is the mission computer. The main properties to enforce are that (i) only

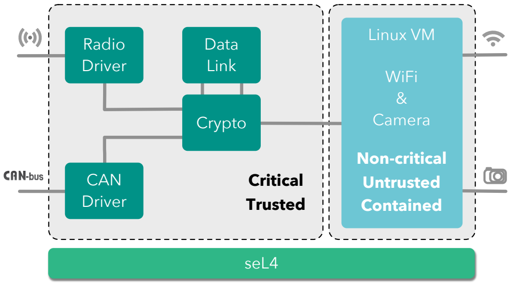


Figure 4: Simplified quadcopter mission computer architecture

correctly authenticated commands from the ground station are sent to the flight computer, that (ii) cryptographic keys are not leaked, that (iii) no additional messages are sent to the flight computer, and that (iv) untrusted payload software cannot influence the flight behaviour of the vehicle. The operating assumption is that the camera is untrusted and potentially compromised (i.e., malicious), that its drivers and the legacy payload software are potentially compromised, and that any outside communication is potentially compromised. For the purpose of this example, we assume a correct and strong cryptography implementation (i.e. 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 4 shows how the quadcopter architecture achieves 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) authorised messages to the flight computer via the CAN interface stack, and to 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 to purely the isolation properties and information-flow behaviour of the architecture, and the behaviour of the single trusted crypto component. Assuming correct behaviour of that component, keys cannot be leaked, because 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, because 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 only communicate to the rest of the system via the trusted crypto component.

It is easy to imagine that this kind of architecture analysis could be automated to a high degree by model checking and higher-level mechanised reasoning tools. As in MILS systems [Alves-Foss et al., 2006], the observation is that 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 behaviour 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 we summarised in Section 3 can achieve this enforcement, but the level of abstraction of these 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, which are created programmatically, and errors in configuration could lead to security violations.

The next section shows how we not only automate the configuration and construction of such code, but also how we can automatically prove that architecture boundaries are enforced.

5. VERIFIED COMPONENTISATION

5.1 Generated Code

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 [Kuz et al., 2007] that runs on seL4 abstracts over the low-level kernel mechanisms, and provides communication primitives as well as support for decomposing a system into functional units, as seen in Figure 5. Using this platform, we can design and build seL4-based systems in terms of high-level *components* that communicate with each other and with hardware devices using *connectors*, such as *remote procedure calls* (RPC), *dataports* and *events*. Internally, CAmkES implements these abstractions using seL4's low-level kernel objects: components comprise (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. Similarly, a dataport connector is implemented using shared memory (shared frame objects present in the address spaces of two components, optionally restricting the direction of communication). Finally an event connector is implemented using seL4's notification mechanism.

CAmkES also generates, in the capDL language [Kuz et al., 2010], 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 initialiser that runs as the first task after boot and performs the necessary seL4 operations to instantiate and initialise the system [Boyton et al., 2013].

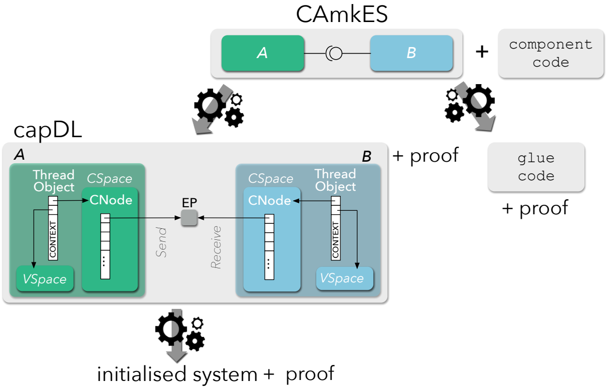


Figure 5: CAmkES workflow

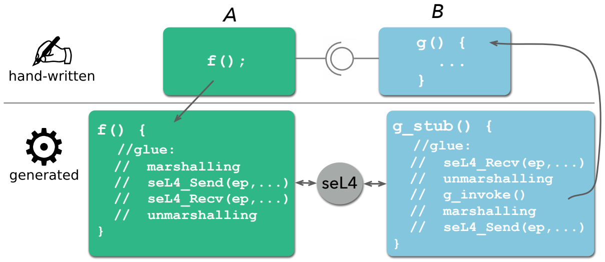
In summary, a component platform provides free code. The component architecture describes a set of boxes and arrows, and the implementation task is reduced to filling in these boxes; the platform generates the rest, enforcing the architecture.

With a traditional component platform, this would mean that the generated code increases the trusted computing base of the system, because it has the ability to affect the functionality of the components. However, CAmkES also generates proofs.

5.2 Automated Proofs

While generating glue code, CAmkES produces formal proofs in Isabelle/HOL, following a *translation validation* approach [Pnueli et al., 1998], demonstrating that the generated glue code obeys a high-level specification, and that the generated capDL specification is a correct refinement of the CAmkES description [Fernandez, 2016]. We also prove that the generic seL4 initialiser 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.

Developers rarely look at the output of code generators, focussing instead on the functionality and business logic of their system. In the same way, we intend the glue code proofs to be artefacts that do not need to be examined, the developer can focus on proving the correctness of their hand-written code. Mirroring the way in which a header generated by CAmkES gives the developer an API for the generated code, the top-level generated lemma statements produce a *proof API*. These lemmas describe the expected behaviour of the connectors. In the example of RPC glue code depicted 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 we generate states that the invocation of the generated RPC glue code f behaves as a direct invocation of g, as if it were colocated with the caller.



```
f();
```

```
g() {
  ...
}
```

```
f() {
  //glue:
  //  marshalling
  //  seL4_Send(ep,...)
  //  seL4_Recv(ep,...)
  //  unmarshalling
}
```

```
g_stub() {
  //glue:
  //  seL4_Recv(ep,...)
  //  unmarshalling
  //  g_invoke()
  //  marshalling
  //  seL4_Send(ep,...)
}
```

Figure 6: RPC generated code

To be useful, the proofs we generate must be composable with (almost) arbitrary user-provided proofs, both of the function g and of the contexts where g and f are used. For this, the specification of the connectors is parameterised by user-provided specifications of remote functions. In this way, the proof engineer can reason about her architecture, providing specifications and proofs for her components, and relying on specifications for the generated code.

To date we have demonstrated this process end-to-end using a specific CAmkES RPC connector [Fernandez, 2016; Fernandez et al., 2015]. Extending the proof generator to support other connectors, allowing the construction of more diverse verified systems, should be simpler to achieve, because other connector patterns (dataports and events) are significantly less complex than RPC.

Next to communication code, CAmkES produces the initial access control configuration that is supposed to enforce architecture boundaries. To prove that 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 [Sewell et al., 2011] mentioned in Section 3.2 to infer authority between objects from a capDL description to lift reasoning to a policy level. To complement this, we have defined rules for inferring authority between *components* in a CAmkES description. The produced proof states that the capDL objects, when represented as an authority graph with objects grouped per component, have the same inter-group edges as the equivalent graph between CAmkES components [Fernandez, 2016]. Intuitively this means that 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 in Section 3.2.

Finally, to prove correct initialisation, we use the generic initialiser that will run as the first user task after boot time. In seL4, this first (and unique) user task has access to all available memory. It uses it to create objects and capabilities according to the detailed capDL description that it takes as input. We proved that the state after execution of the initialiser satisfies the one described in the given specification [Boyton et al., 2013]. Currently, this proof holds for a precise model of the initialiser, but not yet at the implementation level. Compared to the depth of the rest of the proof chain, this 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.

6. SEISMIC SECURITY RETROFIT

In practice there are few opportunities for engineering a system from scratch for security, so the ability to retrofit for security is crucial. Our framework supports this by an iterative process we call *seismic security retrofit*, in analogy to retrofitting existing buildings for more 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 show all details.

The original vehicle architecture is the same as in Figure 3. Its functionality is split over two separate computers: a flight computer, which controls the actual flying, and the mission computer, which performs high-level tasks such as ground station communication and camera-based navigation. The original version of the mission computer is a monolithic software application that runs on Linux. The rest of the example will concentrate 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.

6.1 Step 1: Virtualisation

The first step is to take the system as is and run it in a virtual machine (VM) on top of a secure hypervisor (Figure 7). In the seismic retrofit metaphor, this corresponds to 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 OS, in this case Linux. The kernel provides abstractions of the virtualisation hardware, while the VMM manages these for the VM. The seL4 kernel constrains not only the guest, but also the VMM, so that 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 the system configuration the VM may get access to hardware devices through para-virtualised drivers, pass-through drivers, or both. In the case of pass-through drivers, we 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 does not add any additional security or reliability benefits. Instead, the reason for this first step is to enable step two.

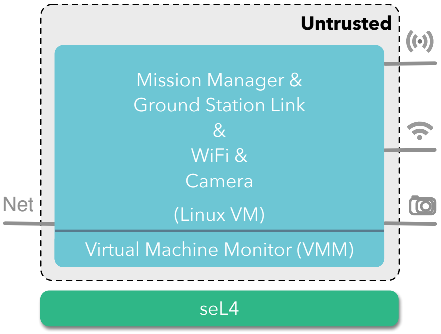


Figure 7: All functionality in a single VM

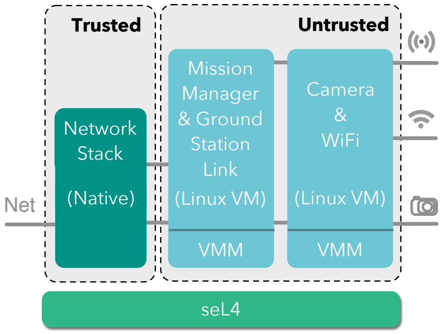


Figure 8: Functionality split into multiple VMs

6.2 Step 2: Multiple Virtual Machines

The second step in a seismic retrofit strengthens existing walls. In software, we improve security and reliability by splitting the original system into multiple sub-system partitions, each of which consists of a VM running the code of only part of the original system. Each VM/VMM combination runs in a separate CAmkES component, which 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 will follow the existing software architecture. However, where the software architecture is inadequate to provide effective isolation, a redesign may be necessary.

Despite the desire for isolation, in general the partitions will need to communicate with each other, so in this step we also add appropriate communication channels between them. It is critically important for security that these interfaces are kept narrow, limiting the communication between partitions to only what is necessary. Furthermore, interface protocols should be efficient, i.e., keeping the required number of messages or amount of data copying minimal. Importantly seL4's ability to enable controlled and limited sharing of memory between partitions allows us to minimise 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, for example, the network interface.

We can iterate this entire step two 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 (Figure 8).

6.3 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 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 reduced footprint and better performance, removing the guest OS and removing the execution and communication overhead of the VMM.

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 hand-written code, through cogeneration of code and proofs, application of model checking, using type-safe programming languages, to static analysis or traditional thorough testing of a smaller code base.

Due to the isolation provided by seL4 and the componentised 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 (Figure 9).

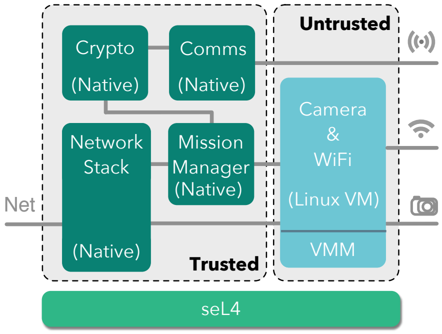


Figure 9: Functionality in native components

The reason for this choice was a trade-off between effort to reimplement the sub-systems and the benefit gained by making them native both from a performance and assurance perspective.

6.4 Step 4: Overall Assurance

With all the parts in place, the final step is to analyse the assurance of the overall system, based on the assurance we can gain from the architecture and individual components.

In HACMS, the communication, cryptography, and mission manager functionality were implemented in a provably type-safe, domain-specific language, Ivory [Elliott et al., 2015] with fixed heap memory allocation. Without further verification, this does not give us high assurance of functional correctness, but it does give us assurance about robustness and crash-safety. Given component isolation, we can reason that this assurance is preserved in the presence of untrusted components such as the camera VM.

The networking component is standard C code consisting of custom code for the platform, and pre-existing library code. Its assurance level is careful implementation and known code. As above, robustness could be increased without much cost using techniques such as driver synthesis [Ryzhyk et al., 2009] and type-safe languages. However, in the overall security analysis of the system, any compromise of the network component would only be able to inject or modify network packets, and since the traffic is encrypted this would not compromise the property that only authorised 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, being isolated, if an attacker were to compromise this VM, she 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, it is important for the mission manager to validate data it receives from the camera VM.

This is the part of system on the ULB that demonstrated containment of a compromise in the in-flight attack mentioned in Section 1. This was a white-box attack, where the Red Team had access to all code and documentation, as well as all external communication, and was intentionally given root access to the camera VM (simulating a successful attack against legacy software). This served to validate the strength of the security claims and to uncover any missed assumptions, interface issues, or other security aspects that the research team might have failed to take into account.

7. LIMITATIONS AND FUTURE WORK

The previous sections have given an overview of a method for achieving high assurance for systems whose security property can be enforced by their component architecture. We have proved theorems for the kernel level and its correct configuration, as well as theorems that 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 currently 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 this 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 our example, it is clearly limited by the fact that it can only express 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 behaviour 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. These could include security patterns such as input sanitisers, output filters, down-graders, and run-time monitors, potentially generated from higher-level specifications, but also infrastructure components such as re-usable crypto modules, key storage, file systems, network stacks, and high-assurance drivers. If the security property depends on more than one such component, we will need 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, there are now real high-assurance systems that we can construct rapidly and with a cost that is lower than traditional testing.

Acknowledgements

We are grateful to Kathleen Fisher, John Launchbury, and Raymond Richards for their support as program managers in HACMS, and in particular to Kathleen Fisher for the vision to start this program. John Launchbury coined the term *seismic security retrofit*. We thank Lee Pike for feedback on a draft of this paper. 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 OS aspects in this paper, the rapid construction of high-assurance systems includes many further aspects, such as a trusted build, trusted components, as well as architecture and security analysis tools.

This material is based on research sponsored by Air Force Research Laboratory and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-12-9-0179. The U.S. Government is authorised 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 Air Force Research Laboratory, the Defense Advanced Research Projects Agency or the U.S. Government.

References

Jim Alves-Foss, Paul W. Oman, Carol Taylor, and Scott Harrison. The MILS architecture for high-assurance embedded systems. *Int. J. Emb. Syst.*, 2:239–247, 2006.

Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *RTSS*, pages 339–348, Nov 2011.

Boeing. Unmanned Little Bird H-6U. http://www.boeing.com/defense/unmanned-little-bird-h-6u/. Visited: October 2016.

Andrew Boyton, June Andronick, Callum Bannister, Matthew Fernandez, Xin Gao, David Greenaway, Gerwin Klein, Corey Lewis, and Thomas Sewell. Formally verified system initialisation. In *15th ICFEM*, pages 70–85, Oct 2013.

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare logic for certifying the FSCQ file system. In *SOSP*, pages 18–37, Oct 2015.

David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In *21st TPHOLs*, pages 167–182, Aug 2008.

David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of some timing channels on seL4. In *CCS*, pages 570–581, Nov 2014.

Ed Colbert and Barry Boehm. Cost estimation for secure software & systems. ISPA / SCEA 2008 Joint International Conference, Technical Report usc-csse-2008-811, University of Southern California, May 2008.

Jared Davis and Magnus O. Myreen. The reflective Milawa theorem prover is sound (down to the machine code that runs it). *JAR*, 55(2):117–183, 2015.

Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *CACM*, 9:143–155, 1966.

Trevor Elliott, Lee Pike, Simon Winwood, Pat Hickey, James Bielman, Jamey Sharp, Eric Seidel, and John Launchbury. Guilt free Ivory. In *2015 ACM SIGPLAN Symp. Haskell*, pages 189–200, 2015.

Matthew Fernandez. *Formal Verification of a Component Platform*. PhD thesis, UNSW Computer Science & Engineering, Jul 2016.

Matthew Fernandez, June Andronick, Gerwin Klein, and Ihor Kuz. Automated verification of RPC stub code. In *International Symposium on Formal Methods*, pages 273–290, Jun 2015.

Robert W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19:19–32, 1967.

Georges Gonthier. A computer-checked proof of the four colour theorem. http://research.microsoft.com/en-us/people/gonthier/4colproof.pdf, 2005.

Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the Odd Order Theorem. In *4th ITP*, volume 7998 of *LNCS*, pages 163–179, 2013.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *OSDI*, Nov 2016.

Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture. *CoRR*, abs/1501.02155, 2015. URL http://arxiv.org/abs/1501.02155.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *25th SOSP*, pages 1–17, Oct 2015.

Gernot Heiser and Kevin Elphinstone. L4 microkernels: The lessons from 20 years of research and deployment. *Trans. Comp. Syst.*, 34(1):1:1–1:29, Apr 2016.

Sudeep Kanav, Peter Lammich, and Andrei Popescu. A conference management system with verified document confidentiality. In *CAV*, volume 8559 of *LNCS*, pages 167–183, 2014.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220, Oct 2009.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *Trans. Comp. Syst.*, 32(1):2:1–2:70, Feb 2014.

Ramana Kumar, Magnus Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *POPL*, pages 179–191, Jan 2014.

Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. Self-formalisation of higher-order logic — semantics, soundness, and a verified implementation. *JAR*, 56(3):221–259, 2016.

Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. 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):687–699, May 2007.

Ihor Kuz, Gerwin Klein, Corey Lewis, and Adam Christopher Walker. capDL: A language for describing capability-based systems. In *APSys*, pages 31–35, Aug 2010.

Xavier Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *S&P*, pages 415–429, May 2013.

Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *4th TACAS*, pages 151–166, Mar 1998.

John Rushby. Design and verification of secure systems. In *SOSP*, pages 12–21, Dec 1981.

Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with Termite. In *SOSP*, pages 73–86, Oct 2009.

seL4. The seL4 microkernel code and proofs. https://github.com/seL4/.

Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In *ITP*, pages 325–340, Aug 2011.

Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *PLDI*, pages 471–481, Jun 2013.

Thomas Sewell, Felix Kam, and Gernot Heiser. Complete, high-assurance determination of loop bounds and infeasible paths for WCET analysis. In *RTAS*, Apr 2016.

Sidebar: Proof Effort

The design and code development of seL4 took 2 person years (py). Adding up all seL4-specific proofs over the years, comes to a total of 18 py for 8,700 lines of C code. In comparison, L4Ka::Pistachio, another microkernel in the L4 family, comparable in size to seL4, took 6 py 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 [2008], a traditional Common Criteria EAL7 certification for 8,700 lines of C code would take more than 45.9 py. That means, formal binary-level implementation verification is already more than a factor of 2.3 cheaper than the highest certification level of Common Criteria, and provides significantly stronger evidence.

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

Appendix

This appendix gives a flavour of the formalisms used in the seL4 and CAmkES proofs, each with a reference to its full description. The statements are slightly simplified to fit them here.

The Functional Correctness Proof

The functional correctness proof links the abstract specification of seL4 to its C implementation, which is later compiled to a verified binary [Sewell et al., 2013]. The following snippet gives a flavour of the monadic functional style of the abstract specification. It shows (simplified) Isabelle/HOL code for the scheduler at the abstract level:

```
schedule ≡ do
  threads ← all_active_tcbs;
  thread ← select threads;
  switch_to_thread thread
od OR switch_to_idle_thread
```

The corresponding efficient C scheduler is spread out over multiple functions and hundreds of lines of code. The technique we use to express functional correctness formally is refinement. Program C *refines* program A (written $A \sqsubseteq C$), if the behaviours of C are a subset of the behaviours of A. We extended this classical notion to state monads using a variant of forward simulation, which we call *correspondence*:

$$\begin{array}{l} \mathit{corres}\ R\ P\ P'\ A\ C \equiv \forall\,(s, s') {\in} \mathit{state\_rel}.\ P\ s \wedge P'\ s' \longrightarrow \\ \quad (\forall\,(r', t') {\in} \mathit{fst}\ (C\ s').\ \exists\,(r, t) {\in} \mathit{fst}\ (A\ s).\ (t, t') \in \mathit{state\_rel} \wedge R\ r\ r') \\ \quad \wedge\ (\mathit{snd}\ (C\ s') \longrightarrow \mathit{snd}\ (A\ s)) \end{array}$$

where A and C are abstract and concrete monadic programs, P and P' are preconditions for the abstract and concrete state, R is a relation between the return values of the monad, and state_rel the refinement relation between abstract and concrete states. Details can be found in [Cock et al., 2008]. The preconditions make this notion contextual and allow us to construct a scalable calculus with moderate automation. After applying this calculus to the entire kernel over three specification levels, we get as a formal corollary the original notion of refinement and arrive at the simple looking theorem that the machine $\mathcal{M}_C$ with the C kernel refines the abstract machine $\mathcal{M}_A$ for any user program u:

THEOREM 1. $\mathcal{M}_A\ u \sqsubseteq \mathcal{M}_C\ u$

Details on this theorem can be found in [Klein et al., 2014].

Integrity

The power of this refinement statement is that it allows us to derive Hoare triples about $\mathcal{M}_C$ by deriving Hoare triples about $\mathcal{M}_A$. The integrity property of Section 3.2 is such a Hoare triple.

The formalisation builds on the predicate $\mathsf{pas\_refined}(p, s)$, which captures authority confinement: the authority of each subject in the system state s does not exceed its authority in the policy p.

$$\begin{array}{l} \mathsf{pas\_refined}(p, s) \equiv \\ \mathsf{policy\_wf}\ (\mathsf{policy}\ p)\ (\mathsf{range}(\mathsf{IRQAbs}\ p))\ (\mathsf{subject}\ p) \wedge \\ \mathsf{irq\_map\_wf}\ p\ s \wedge \\ \mathsf{auth\_graph\_map}\ (\mathsf{objectAbs}\ p)\ (\mathsf{objs\_to\_policy}\ s) \subseteq \mathsf{policy}\ p \wedge \\ \mathsf{asids\_to\_policy}\ p\ s \subseteq \mathsf{policy}\ p \wedge \\ \mathsf{irqs\_to\_policy}\ p\ s \subseteq \mathsf{policy}\ p \end{array}$$

Sewell et al. [2011] describe the details of this definition.

The integrity theorem captures the allowed modifications $\mathsf{integrity}(p, s, s')$ between the states s and s' before and after any kernel call, according to a policy p. The relation is transitive and reflexive in s and s'. We can now express the integrity property of Section 3.2 as a Hoare triple:

$$\begin{array}{l} \{\!|\lambda s.\ \mathsf{pas\_refined}(p, s) \wedge \mathsf{invs} \wedge (ev \neq \mathsf{Interrupt} \longrightarrow \mathsf{ct\_active}) \wedge \\ \ \mathsf{is\_subject}\ p \circ \mathsf{cur\_thread} \wedge (s = st)|\!\} \\ \quad \mathsf{call\_kernel}\ ev \\ \{\!|\lambda s.\ \mathsf{integrity}(p, st, s)|\!\} \end{array}$$

where call_kernel is the top-level abstract kernel function and invs are the kernel invariants. The other preconditions are explained in [Sewell et al., 2011]. With refinement, we get the same Hoare triple for free for the C code.

CapDL and System Initialisation

Section 5 mentions capDL specifications that describe the authority state of a system at boot time, in particular which objects should exist and which capabilities they possess. Formally, this is a further abstraction $\mathcal{M}_\mathcal{D}$ over the abstract functional specification, with a further refinement theorem $\mathcal{M}_\mathcal{D}\ u \sqsubseteq \mathcal{M}_A\ u$.

The specification of the user-level component init_system uses kernel calls in $\mathcal{M}_\mathcal{D}$, which allow us to prove that wellformed capDL specifications lead to correctly initialised kernel objects:

THEOREM 2. *If $\mathit{well\_formed}\ spec$ and $obj\_ids = \mathit{dom}\ (\mathit{cdl\_objects}\ spec)$ and $\mathit{distinct}\ obj\_ids$ then*

$$\begin{array}{l} \{\!|\text{«}\mathit{valid\_boot\_info}\ bootinfo\ spec \wedge^* R\text{»}|\!\} \\ \mathit{init\_system}\ spec\ bootinfo\ obj\_ids \\ \{\!|\lambda s.\ \exists\,\varphi.\ \text{«}\bigwedge^* \mathit{map}\ (\mathit{object\_initialised}\ spec\ \varphi)\ obj\_ids \wedge^* \\ \qquad\qquad \mathit{si\_objects}\ spec\ \varphi \wedge^* R\text{»}\ s \wedge \\ \qquad\qquad \mathit{injective}\ \varphi \wedge \mathit{dom}\ \varphi = \mathit{obj\_ids}|\!\} \end{array}$$

Boyton et al. [2013] explain the predicates and separation logic notation in this statement and [Klein et al., 2014] prove an additional connection directly to the integrity statement.

Component Separation

The authority in a system is governed by an access-control policy p, in the form of a directed graph. The CAmkES component system generates such a policy from the input system architecture. The authority graph for a CAmkES system contains a node for each component and connection in the system, with the edges between them labelled with the authority used to implement the connection.

The generated policy contains one node for every component and connection, which means the engineer has the flexibility to further group components and connections into subsystems according to her desired security domains. Although this graph can become large for a non-trivial system, reasoning support on this level is fully automatic. For example, it is possible to state an authority confinement claim as a set membership property, such as the one shown in Theorem 3 below, and decide it automatically.

THEOREM 3. $(agent_A,\ \mathit{Reset},\ agent_B) \notin \mathit{auth\_graph}\ p$

The level of abstraction of this automatic reasoning is at the granularity of CAmkES components. In addition, we know that the capDL specification CAmkES generates implements this policy correctly:

THEOREM 4. $\mathit{CAmkES\_gen}\ spec\ ext\ irqs = \mathit{Some}\ (capdl, \mathcal{P}) \wedge p \in \mathcal{P} \Longrightarrow \mathit{pcs\_refined}\ p\ capdl$

where pcs_refined is pas_refined from above, expressed directly on the capDL level. See [Klein et al., 2014] for the formal connection.

[Fernandez, 2016, Chapter 7] provides the details of this theorem, and the assumptions CAmkES_gen entails.

This means, together with the integrity and refinement theorems, and the binary verification theorem (not shown here), this set of theorems spans the entire chain from the CAmkES component level down to the binary, without any proof input required from the engineer who uses CAmkES.

The proofs are either already provided, generated or automatic.