SSC17-XII-08

Cyber in a World of Plenty: Secure High-Performance On-Orbit Processing[1]

Kyle Ingols, Jeff Brandon, Eric Koziel, Michael Zhivich
244 Wood Street, Lexington MA 02421
kwi@ll.mit.edu

ABSTRACT

The days of "dumb" satellites in LEO are numbered. As many CubeSat missions have proven, commercial off the shelf (COTS) processors – orders of magnitude more powerful than traditional rad-hard parts – can fly. Powerful processors give satellite designers the horsepower they need to collect, analyze, and process big datasets on-orbit. The extra headroom also accommodates rapid development using traditionally-terrestrial COTS and open-source operating systems and software stacks. Unfortunately, these large software ecosystems bring their terrestrial cyber sins into orbit with them. We need to understand and mitigate the cyber threat now, before bad patterns become entrenched and propagated.

This paper reviews a set of cybersecurity guidelines that help developers craft more securable designs for small satellites. The guidelines highlight ways that satellite security and ground system security can strengthen each other. Since system compromise remains possible even if best practices are followed, the guidelines suggest ways to recover control. We further describe ongoing work in a reference implementation that honors the guidelines, building on the seL4 microkernel as the security foundation and NASA's Core Flight Software (cFS) as the functionality foundation.

[1] DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.

INTRODUCTION

Satellites have historically had modest amounts of computational power. Radiation-hardened parts with extensive flight heritage are naturally the components of choice for traditional, large, expensive satellite systems. These expensive, low processing power systems had correspondingly straightforward software loads.

The SmallSat paradigm – especially the LEO CubeSat regime – is substantially changing the calculus in space. CubeSats willing to tolerate the higher risk of COTS components can fly tremendous amounts of computational power for a fraction of the cost and size, weight, and power (SWaP) of traditional rad-hard components. This has opened up a world of additional possibilities for on-orbit data processing.

However, the additional headroom has also made it possible to use traditional COTS software in space as well. This is a difficult temptation to resist: the rapid development timelines of small satellites would certainly benefit from the use of COTS technology typically seen in the embedded computing space. It is no longer unheard of to fly Linux in space, for example, with all of the overhead of a traditional distribution. Such an approach brings with it all of the cyber security risks of these large, expansive computing environments.

It's difficult enough to survive with this level of vulnerability on the ground; it becomes nearly untenable in space. A space-based computer has only itself as its root of recovery; no trusted human can come along and push the reset button, or unplug it, or replace the hard drive. An adversary could conceivably compromise a satellite processor to such a degree that the rightful owner permanently and irrevocably loses control of their satellite.

We can't simply return to the era of low power processing, either; the increased horsepower enables novel on-orbit processing approaches that are too valuable to abandon. We must find some middle ground, a way to have robust processing capability on orbit coupled with the safety and security guarantees necessary to ensure mission success and recoverability in a challenging world. This paper documents our efforts to achieve that goal.

If our ground control system is impervious to attack, and our vendors provide only malware-free software for our use, and our authorized administrators never misbehave, then the attack surface is minimal. However, reality is often less assured. We have selected an attack surface that assumes temporary loss of control of the ground control system, allowing an adversary to issue properly authenticated commands and software updates to the satellite for a brief time. The way that the adversary is detected and evicted from the ground control station, while important, is beyond the scope of this paper. Our goal is to make such a transient compromise non-fatal for the satellite.

We have established a set of *guidelines* for design of space systems that move beyond the basic and traditional security approaches of encrypted command-and-control links to consider the broader set of threats to space systems, the residual threats faced even if encrypted C2 is used. We review these guidelines in the next section. They receive a more thorough treatment in [2].

Additionally, we have been developing an example implementation of secure satellite processor, with those guidelines in mind. The bulk of the paper reviews this effort and reports on our ongoing results and planned future work.

GUIDELINES

To mitigate those three scenarios, we propose design guidelines geared to motivate more secure and more securable designs. We deliberately neglect traditional security approaches one might otherwise employ from, e.g., NIST controls **[3]**, as they are adequately covered there.

In [2] we call out five key guidelines, which we summarize here.

First, **fail slowly** – ensure that any dangerous commands that your satellite can receive have a substantial margin between when the command is received and when the command is obeyed. This gives ground controllers time to recognize and countermand if the command is unintended.

Second, employ **crypto beyond COMSEC**, to cryptographically enforce role-based access control. This provides finer grained security vs. the basic approach of offering complete trust to any ground control entity on the good side of the encryptor.

Third, and most importantly, satellites must provide a **root of recovery**. In traditional, terrestrial systems, any amount of compromise can effectively be remediated by a *human being* asserting physical control – rebooting, reimaging, or replacing the compromised system. For satellites, that is a very expensive and slow luxury. A root of recovery becomes imperative.

In [2] we also consider **ablative defenses** as a technique to reduce the risk that the security

enhancements themselves will jeopardize the mission, but we do not consider it further here.

Lastly, we wish to **reboot and succeed quickly**. If malware must be evicted by restarting the satellite, the restart sequence should be as short as possible. Ideally, restart takes *seconds* and allows controllers ample time to recover the satellite and resume operations. This also has benefits for SEE mitigation, which we revisit later in the paper.

Fast reboot is not always the answer. If the system ends up in a loop where it reboots very frequently, perhaps a more thorough hardware self-test is warranted. The satellite can also differentiate between "soft" (software-initiated) and "hard" (watchdog-initiated) reboots and perform self-tests accordingly.

KEY COMPONENTS

We are developing an example space processing environment that is a "one size fits many" model. Our ultimate goal is to provide an environment that developers can use almost as readily as the less secure options, so that a superior security foundation can be had even within the aggressive timelines of a traditional small satellite effort.

We have selected a set of representative software and hardware. For the underlying processor, we focus on the Zynq 7000 series parts, used in the CHREC Space Processor (CSP) and other products. The Zynq is a very powerful and capable system-on-chip (SoC), featuring two ARM cores for general-purpose processing, FPGA configurable logic, and an ARM AXI bus to connect the two. The SoC additionally offers easy support for a variety of hardware interfaces, including Ethernet, I2C, SPI, and GPIO pins.

The operating system is a crucial underpinning to the ultimate security of the system. Rather than a traditional OS choice like Linux or even FreeRTOS, we are using seL4. The seL4, or "security enhanced L4," microkernel provides unparalleled safety and security at the OS level [4]. It is *correct by construction* – the seL4 specification is written in a machine-readable language, and formal methods are used to reduce the specification to an *implementation* that is proven to implement the specification faithfully. Large classes of vulnerabilities – buffer overflows, use-after-frees, and the like – simply don't exist.

An OS that provides such strong separation may offer other benefits beyond traditional security. It could make it safe to co-host bus, payload, and cryptographic processing in the same processor, further saving SWaP on orbit.

Although one could imagine building directly on seL4 – and we do just that for some things – we can better serve the community if we can support an environment with which they are already familiar. We are investigating the feasibility of using NASA's Core Flight Software (cFS) as our representative software stack. cFS includes an operating-system abstraction layer (OSAL) for easier portability between operating systems. Later in the paper we describe our experiences porting OSAL to seL4.

Lastly, we require a test infrastructure within which we can verify that our processing environment operates properly and supports satellite operations. We intend to use the NASA Operational Simulator for Small Satellites (NOS3) satellite simulation environment for this purpose. NOS3 includes a physical model of orbit, a ground station, and key pieces of emulated satellite hardware (EPS, radio, etc.).

The NOS3 emulation environment assumes the use of the Linux operating system, however, and cannot simulate satellite software running on some other OS such as FreeRTOS or seL4. The NASA IV&V group is actively developing extensions to NOS3 that decouple the NOS3 simulation environment from the satellite control system itself, instead connecting the two only via the real, physical interfaces by which data flows on orbit (I2C and SPI in this case). When complete, NOS3 will be able to drive arbitrary C2 systems on arbitrary processing hardware using NOS3's emulated peripherals.

KERNEL

The seL4 microkernel is the result of work done via NICTA and the DARPA "High-Assurance Cyber Military Systems" (HACMS) program. The work consisted of two staff-years building the actual microkernel, and twenty staff-years proving that the microkernel is correct. The result is perhaps the most thoroughly vetted 8,000 lines of C code in the world.

Although it substantially advances the state of the art, the formal methods pedigree of seL4 is not in itself a panacea. seL4 is guaranteed to faithfully implement its specification, *if* the proof tools are correct, and – more importantly – *if the specification is correct.* The specification is written by humans, and humans have not been proven to be correct. The hope is that the specification is a far simpler and easier thing to get right than the actual nuts and bolts needed to implement it. Regardless of those limitations, however, seL4 offers guarantees well beyond those of heavier-weight operating systems such as Linux.

Compared to Linux, however, seL4 is much harder to work with. It doesn't have a robust ecosystem yet, it doesn't lend itself well to GUIs or enterprise remote administration tools or even common IDEs, and setting up an execution environment around it is challenging.

For all those challenges, however, seL4 is still a viable choice for embedded processing. One need only manage I/O to on-board systems via device drivers; no complicated UI support is needed. Although administrators won't have the easy familiarity of shell access to poke around their satellites, seL4 should be more than capable of handling more traditional command and control (C2) frames.

API

Compared to the hundreds of system calls supported by Linux, the seL4 kernel has a remarkably svelte seven:

```
seL4_Send()
seL4_NBSend()
seL4_Call()
seL4_Wait()
seL4_Reply()
seL4_ReplyWait()
seL4_Yield()
```

These seven calls comprise a message-passing paradigm by which distinct processes can communicate. At first blush this seems to be insufficient; it is unclear how one establishes the actual processes that are communicating via this API in the first place.

In fact, seL4 hides a lot of complexity (and a series of other calls) in the concept of *capabilities*. A capability-based scheme allocates specific rights to specific processes – message queues that they are allowed to access, memory regions that they are allowed to use, the right to make another process, and so on – and gives those processes the ability to delegate privileges to other processes. At system startup, a single "init process" is given full rights to every capability in the system, and this process spawns the additional processes and delegates the credentials as needed to accomplish the system's goals.

Processes in seL4 communicate with each other and with the kernel using *inter-process communication* (IPC) message passing. A process or thread capable of sending or receiving a given message on a given IPC channel is called an *endpoint*. seL4 uses these extensively to pass capabilities as well as process-specific data.

seL4 includes in its ecosystem several other tools that aid in compile-time configuration and runtime interaction. Chief among them is the component architecture for microkernel-based embedded systems (CAmkES). The CAmkES system allows developers to define a static system configuration at *compile time*, indicating which processes are started in which order and communicate via which means with which other processes and so on. CAmkES then auto-generates the seL4 calls necessary to establish the identified scheme.

The CAmkES toolchain is not yet formally verified, but it is on the development path. This becomes a critical component of the ultimate root of recovery, as the configuration of seL4 must be robust enough to ensure that the root of recovery is appropriately privileged and isolated relative to the system it is expected to recover.

Process Management

Dynamic allocation and management of resources in seL4 is possible, but hard to do. It is far easier if every process, mutex, semaphore, IPC channel, and the like are known *a priori* and configured appropriately. We believe this is a plausible model for spacecraft processing.

However, we must also contend with recovery. If an individual process in seL4 goes awry, how can we identify this anomaly and recover from it in a safe and predictable manner?

We plan to address this problem with another process, the *respawn daemon*, or `respawnd`. This daemon is responsible for monitoring existing processes for bad behavior – or, more specifically, for receiving *signals* (just another IPC message) from the seL4 kernel if the process has done something untoward. The respawn daemon can then restart the process as appropriate.

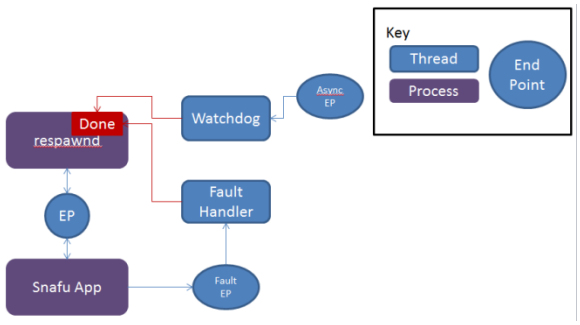

Figure 1. IPC channels between `respawnd` threads and the SNAFU process. The watchdog and fault handler threads are responsible for alerting the `respawnd` process if SNAFU fails.

As Figure 1 shows, `respawnd` serves as the endpoint for messages about a process's misbehavior. We use, for testing, a simple "echo server" process to provide IPC traffic to and from a "SNAFU server" process.

The SNAFU server is designed to deliberately misbehave – hang, or attempt boundless memory allocation, for example – so that we can test the `respawnd` process's ability to detect and react to a problem. This work is ongoing.

HARDWARE SUPPORT

No software is an island. A satellite processor must be able to receive information from the various peripherals that comprise a satellite, and must be able to command them as appropriate to carry out the satellite's mission and to ensure satellite survival. This outreach necessitates support for device drivers.

We are focusing initially on I2C- and SPI-based devices, as these are common busses by which a variety of peripherals communicate. Importantly for us, these are the two busses supported by the NOS3 environment, giving us a clear means to test what we develop.

We are beginning our work with the I2C protocol, with the near-term goal of performing proper C2 on the NOS3 emulated EPS.

I2C

The Inter-Integrated Circuit (I2C) is a two wire protocol intended for short distance communications within a single device. It has a single, fixed bus master, and allows a single bus to connect up to 127 peripheral devices [5]. It supports a range of bandwidths as well, since a single misbehaving peripheral can bring the entire bus down, we plan a single peripheral per bus. The Zynq system will serve as the I2C master for each.

A newer variant called I3C exists, and offers backwards compatibility with I2C coupled with new features and higher data rates. We opted not to use it. In the near- and mid-term it offers no additional value, as the devices we are targeting do not use I3C. Additionally, the I3C standard is not freely available. We see no value in paying the necessary fees to read the standard, much less to try and implement it. .

To get from the seL4 environment to the I2C pins, we need to do a series of tricks. We must allocate the correct memory-mapped I/O (MMIO) memory range to reach the bus itself. We must get this physical address range into a seL4 process by issuing a seL4 capability providing exclusive access to the physical pages. Lastly, we must provide FPGA-based logic to bridge that MMIO access to the actual I2C pins.

The physical address of the I2C device is determined and assigned by the Xilinx SoC design software. (At present, we do not choose the address ourselves.) Once we have this address, we use seL4 utilities to assign capabilities and map that address so that it may be used by a process that serves as a device driver.

There are three components necessary to successfully map a physical address in seL4: a virtual address space or *vspace*, a virtual kernel allocator or *vka*, and an IO mapper.

A *vka* is a seL4 type-aware object allocator that will allocate new kernel objects. It is commonly used in the creation of many seL4 types. It is created from seL4 environment information, memory pool size information, and a reference to a memory pool. A *vspace* is a seL4 virtual memory management tool and is created using a previously instantiated *vka* and static memory for bootstrapping the virtual memory, along with information on the boot info frame so that it may be marked as a reserved region. Vspaces may be shared between threads, or not shared for isolation [7].

An IO mapper is a seL4 tool for mapping physical memory addresses to a vspace. The user creates an IO mapper in the seL4 root process using a vspace and vka. The vka also ensures that capabilities are assigned correctly to the mapped address range so that the calling process has the capability required to read or write to the newly mapped region, which can then be passed along to other seL4 processes. In this case, because the root process has capabilities to everything, the call will succeed when it checks to make sure the calling process has the appropriate capability to access that memory region.

Within seL4, memory mapping is a matter of capabilities management. With these three components in hand, we use a call to the `ps_io_map()` function to map our desired physical address to a virtual address capability so we can use it in a seL4 process. The mapping function uses the IO mapper, the physical address we wish to map, the size of the space we wish to map, a flag indicating whether or not the references should be cached (i.e. Does data need to be written to or read from the device directly each time because either the data is controlling the device or data is from a volatile sensor that may change based on factors external to our process), and flags indicating whether the space should be mapped for reading, writing, or both. As the resulting virtual address range is a capability, like everything else in seL4, we can then provide a device driver process with the necessary rights to reach the page, and deny access to all other processes.

Xilinx provides autogenerated "glue code" to connect the remaining dots. The glue software provides an API

to drive the I2C bus from software, and the glue firmware provides FPGA fabric elements to bridge the memory-mapped IO registers to the controller and physical bus interface pins. The Xilinx-provided software functions directly use the virtual address we provisioned. They work off the base virtual address and use appropriate offsets to exercise the desired parts of a component. We have found that the autogenerated software and firmware work cleanly within seL4, making for an easy and familiar foundation for an I2C driver.

At present, we are performing basic tests using a ZedBoard to host seL4 on Zynq an Aardvark I2C adapter to emulate a virtual peripheral from a computer. In the near future we plan to migrate to the NOS3 EPS emulator as the peripheral under test, and develop a seL4-compatible device driver for it.

CORE FLIGHT SYSTEM

NASA's Core Flight System is a reusable set of basic components and optional applications that allow for efficient construction of a flight software load and easy reuse of components across missions ***[8]***. We selected cFS as our representative layer for two pragmatic reasons: first, because it is widely used, including in the small satellite world; second, because it is freely available.

OSAL

cFS does not target one specific operating system. Instead, it provides an operating system abstraction layer (OSAL) that in principle allows cFS to run on a variety of operating systems. OSAL supports Linux, VxWorks, and FreeRTOS, allowing adopters to choose the OS features they desire.

To use cFS on seL4, we must necessarily provide an adaptation layer for OSAL. This becomes somewhat complicated, however, as OSAL assumes certain POSIX-like features that seL4 simply doesn't present. In this section we describe our exploration of OSAL and our assessment of its portability.

The OSAL API has a total of 99 calls in a handful of broad categories:

- *Miscellaneous* – overall initialization, debug printing, and time measurement
- *Queue* – a reasonably direct translation to seL4 IPC
- *Semaphore and Mutex*
- *Task Control* – creating and destroying tasks, which in OSAL are *threads*, not *processes*
- *Dynamic Loader and Symbol Lookup* – for support of runtime module loading and unloading
- *Timers*
- *Networking* – strangely minimal, with no actual I/O support
- *File System* – a subset of the NFS API, with basics for creating, reading, writing, moving, copying, and deleting files
- *Interrupts*
- *Exceptions*

Luckily, not everything in this API list must be ported, and much of what remains is straightforward. The Exceptions and Networking APIs, for example, are actually never used by cFS, so we ignore them completely. (A smattering of other calls can similarly be neglected.)

The Semaphore and Mutex support will need to be done via IPC, with a dedicated process that manages the resources. CAmkES can support the creation of support for these. Dynamic loading is deliberately neglected; for seL4, we are using static configurations. The File System appears to be used for very small quantities of data – not large sensor outlays or the like – and can be efficiently supported with a simple backing store.

Interrupts, Timers, and Task Control remain a concern. Although the first two should ultimately be surmountable, we expect that a lot of additional work is required to handle OSAL-style task control in the seL4 model. Our initial approach will be to forbid the dynamism that this API reports, instead requiring adopters to use only a statically configured set of processes and interconnects. We hope to ease the configuration by instrumenting OSAL on, e.g., Linux to identify which resources are created and issued, and then simply preallocating them on seL4 and handing them out to the OSAL allocator functions as if they had been dynamically created on the spot.

cFS (and thus OSAL) makes very heavy use of *global variables* as a way to share state. Although convenient and memory-efficient, global variables are vastly less safe and secure than traditional parameter-passed models of data sharing. This idiom was adopted presumably to foster higher speed by minimizing the number of memory-to-memory copies necessary, and when resources were limited in space, it may have been a necessary evil. It is no longer.

To support this global-variables approach, OSAL treats separate "tasks" as *threads*. This means that every task in cFS shares the same memory space. There is no meaningful restriction in access between disparate threads; one can easily read, write, or corrupt data used by others. It is only by extensive testing and the

absence of an adversary that these systems work as well as they do.

To provide enhanced security, we feel it is necessary to migrate cFS Tasks into separate *processes*, not threads, so that there are clear and explicit separators between disparate tasks' privileges. A separate "global variable controller" task would hold the globals and carefully mete out access to them as required.

Although this sounds straightforward, we expect this to be a large level of effort, as we must identify all references to globals, isolate them in getters/setters, and identify what static restrictions can be placed on them. We plan to use the clang C frontend to build a set of pre-processing tools to help us explore the codebase and identify the globals and interdependencies in an automated manner. Even if we succeed, we have not necessarily created a secure system; while we reduce cFS to doing only what it normally does, there is no guarantee that what it normally does is in fact secure.

Although we do not imagine it yet in this effort, there may be merit to considering a more "ground-up" re-engineering of spacecraft C2 software with cyber security and more processing power in mind.

NEXT STEPS

We have made good initial progress on our desired track. We have a ZedBoard for representative testing on the Zynq platform, as shown in Figure 2, and have successfully loaded seL4 onto the ZedBoard for hardware execution and debugging. We have demonstrated a physical memory page mapping scheme for seL4 on Zynq, laying the groundwork for all future hardware driver I/O.

Figure 2: ZedBoard Development Environment

A great deal remains to be done, however, before the system is truly ready to support mission applications. We envision additional work in functionality, reliability, and safety, and all that happens before we truly begin work on the enhanced security systems we aspire to build atop the seL4 foundation.

Hard Realtime Support

Although it is in their development plan, at this time seL4 is not a hard realtime operating system. The maximum execution time of every seL4 system call is bounded and verified, so it is certainly very capable of eventually supporting realtime operations. However, for near- and possibly mid-term use, workarounds may be required.

We plan to use the FPGA fabric to support hard realtime needs, while leaving the ARM cores to perform higher-level planning. Instead of communicating directly to I2C, the ARM core will communicate to a "low-level scheduler" in the FPGA fabric that will queue commands until pre-set times, and then issue them.

Performing the queue-and-execute in the FPGA supports critical timing of "dumb" peripherals. A system that needs to issue a "begin thruster burn" command followed an exact number of milliseconds later by a "cease thruster burn" command, for example, can rely on that C2 channel's dedicated FPGA logic to monitor the time and issue the commands appropriately. In effect, we put a tiny hard realtime processor between each device and the seL4 system, and the seL4 system uses a dedicated protocol between itself and this intermediary.

This does not address every hard realtime issue, however. In the case where information is received from a sensor and the ARM core must process the information and issue an action within a finite amount of time, seL4 as-is may not be fully up to the task. However, if these processing tasks are not particularly complicated, it may again be possible to handle them via extremely simple logic (or, if necessary, code and a soft microcontroller) in the FPGA fabric. Because our first target is an I2C bus with the Zynq as the master, we have adopted a polling model for reading data back from the bus. However, an interrupt model should also be feasible to support with seL4.

AMP Core with Arbiter

Single event effects (SEEs) are the primary Achilles heel of COTS components in space. Unlike rad-hard components, they have no built-in defense to SEEs. The system must instead tolerate errors anywhere from once an hour to once a week, depending on altitude, proximity to the South Atlantic Anomaly, space weather, and good fortune.

The traditional mitigation is to include a small, rad-hard "watchdog" that observes the COTS component and resets the component if anomalous behavior is

observed. This approach can be found on the CHREC Space Processor (CSP) [9] and other products.

The problem is that the watchdog can only reset if there's clear, unmistakable evidence of an upset. If an SEE causes a bit to flip in a cache line somewhere, for example, and it alters a coefficient but doesn't alter any code, the part may continue to operate for quite some time on the bad data, possibly even requiring ground observation and intervention to detect, react, and recover from the SEE.

The Zynq fabric has two options for protecting itself. First, it can implement triple modular redundancy (TMR) to reduce the likelihood that an SEE will corrupt *data*, and it can implement additional logic that periodically scans the SRAM configuration bits for damage so that failures can be detected and remediated rapidly [6]. Remediation could involve actively rewriting SRAM with good values, or (as a safer option) the scan tool may simply restart the part, or ask the watchdog circuit to do so, if an upset is detected.

The ARM cores within the Zynq, however, have no efficient introspection options. A hit to the L2 cache, by far the largest physical feature of the subsystem, would corrupt memory in unpredictable ways. The L2 cache can be disabled to mitigate that risk, but hits to the ALU, register file, or other areas of the ARM cannot be avoided or ignored.

Absent mitigation, the next best option is rapid detection. If the Zynq's two ARM cores were configured to run the same code in parallel, the FPGA fabric could monitor all I/O and ensure that the two cores are always doing exactly the same thing at the same time. Any deviation would indicate an SEU and would be cause for immediate reboot.

Recent research from Sandia National Laboratory [10] has advanced exactly that concept, using the asymmetric multi-processing (AMP) configuration of the Zynq. AMP allows the two cores to operate independently. Unfortunately, there is no clean way to leverage the L2 cache across both cores in an AMP configuration, so there remains a rather substantial performance hit in its absence.

Once an SEE is noticed and the part resets, the onus is on the code to *recover quickly* to an operational state. We will ensure that what we engineer supports this goal, but more importantly we must encourage engineers to keep this in mind when architecting their systems.

Root of Recovery

A secure OS can, by itself, limit an adversary's ability to propagate and persist once some aspect of the user's code is compromised. However, it provides no guarantees that the rightful owner of the satellite can actually evict the adversary, or even command the satellite at all, once a compromise has occurred.

If a critical subset of the satellite's systems – power, a low bandwidth C2 radio, and program EEPROM, for example – can be isolated and formally verified as correct, then that redoubt can be used as a proper root of recovery. The adversary may be able to compromise every other part of the system in some way, but the owner can still use the root of recovery to reassert control, reboot the satellite to a previous (presumably uncompromised) state, and resume operations.

The root of recovery depends on many components. Satellite EEPROM must be properly partitioned and write-protected, so adversarial action can't alter previous states in an unauthorized fashion. The storage device must resist radiation damage. Last but not least, the software components of the root of recovery must be impervious to attack and able to properly reach the radio (so the recovery order can be received), command the EEPROM (for read/write as appropriate), and effect the reboot.

This doesn't necessarily prevent an adversary from denying access to the satellite eventually, however, by orienting the solar panels in a bad direction or the like. However, if that critical subset of impervious code includes sufficient logic to ensure spacecraft survival, then the owner can recover control at their convenience. In addition to the safety benefits, a hard-coded, proven "self-preservation instinct" would ensure that control can be re-asserted regardless of the adversary's software-based actions.

SUMMARY

Satellite systems operate in a very challenging regime, far from their owners, beyond hope of physical repair, and under constant assault by cosmic radiation. Traditional methods of mitigating cyber attack are insufficient in such a challenging environment.

We seek to improve the world of satellite security by laying a foundation that respects the satellite's role as its own root of recovery. A certain subset of the code must simply be immune to compromise, or no useful guarantees can be made. We propose seL4 as one key component of that foundation and are working to lay the groundwork for additional components. The result, if successful, will be a new platform that mitigates much of the cyber risk and allows satellite developers

to focus their attention on the missions they wish to accomplish, rather than on the miscreants that wish to disrupt them.

Secure Communications and Role-Based Access Control

Basic communications security is a necessary prerequisite to a secure satellite system. We would ideally also incorporate more nuanced cryptographically-enforced security, tying specific commands to specific cryptographic principals authorized to issue them. Plotting out those commands, the key management strategy, and the algorithmic and protocol choices is an exciting future direction as well.

Traditionally cryptography is handled in separate, dedicated hardware. If we can build up enough trust in seL4 to separate and protect the cryptographic logic and enforce proper data flows, it may be possible to host a cryptographic implementation, e.g. Lincoln Laboratory's own LOCKMA library [11], within the same processor.

Acknowledgments

We gratefully acknowledge Rob Cunningham and Roger Khazan for their feedback on the paper's structure and content.

References

[1] CVE Details, "CVE Details," [Online]. Available: http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33. [Accessed 7 June 2017].

[2] K. Ingols, "Design for Security: Guidelines for Efficient, Secure Small Satellite Computation," in *Proceedings of the 2017 International Microwave Symposium*, Honolulu, HI, 2017.

[3] Joint Task Force Transformation Initiative Iteragency Working Group, "NIST Special Publication 800-53 Revision 4: Security and Privacy Controls for Federal Information Systems and Organizations," National Institute of Standards and Technology, April 2013.

[4] G. e. a. Klein, "seL4: Formal verification of an OS kernel," *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles,* 2009.

[5] NXP Semiconductors, "UM10204 I2C-bus specification and user manual," NXP Semiconductors, 2014.

[6] A. Jacobs, G. G. A. Cieslewski, A. Gordon-Ross and H. Lam, "Reconfigurable Fault Tolerance: A Comprehensive Framework for Reliable and Adaptive FPGA-Based Space Computing," *ACM Transactions on Reconfigurable Technology and Systems,* vol. 5, no. 4, 2012.

[7] NICTA Trustworthy Systems, "seL4 Reference Manual Version 2.0," 1 December 2015. [Online]. Available: https://sel4.systems/Info/Docs/seL4-manual-2.0.0.pdf. [Accessed 7 June 2017].

[8] D. McComas, *NASA/GSFC's Flight Software Core Flight System,* San Antonio, TX: Workshop on Spacecraft Flight Software, 2012.

[9] D. e. a. Rudolph, "CSP: A multifaceted hybrid architecture for space computing, 2014.

[10] R. Kral and a. et, "Implementation of a Loosely Coupled Lockstep Approach in the Xilinx Zynq7000 All Programmable SoC for High Consequence Applications," in *GOMACtech*, 2017.

[11] R. Khazan and D. Utin, "Lincoln open cryptographic key management architecture," MIT Lincoln Laboratory, Lexington MA, 2012.