Secure Satellite Software-Defined Payloads with High-Assurance Post-Quantum Cryptography

Karthikeyan Bhargavan
Cryspen
Berlin, Germany
karthik@cryspen.com

Thomas Gazagnaire
Parsimoni
Pau, France
thomas@parsimoni.co

Franziskus Kiefer
Cryspen
Berlin, Germany
franziskus@cryspen.com

Virgile Robles
Parsimoni
Pau, France
virgile@parsimoni.co

***Abstract*—As space missions increasingly rely on software-defined payloads, the need for secure, reliable, and future-proof communication becomes critical. This paper presents the integration of formally verified, post-quantum cryptography into SPACEOS, a unikernel-based operating system for satellite platforms. We leverage LIBCRUX, a high-assurance cryptographic library written in Rust, and BERTIE, a verified TLS 1.3 implementation, to enable memory-safe, side-channel resistant, and quantum-secure communication and update mechanisms. This proposed integration enables applications ranging from secure software updates to authenticated channels for quantum key distribution and key establishment for SDLS-based protocols. We demonstrate the viability of this approach with a proof-of-concept implementation for post-quantum signed software updates. This work is a step toward providing robust security foundations for next-generation space systems with minimal developer burden.**

***Index Terms*—Formal Methods, Post-Quantum Cryptography, Unikernels, Virtualization**

I. INTRODUCTION

As small satellite missions grow in complexity, software inefficiencies increasingly hinder performance, security, and scalability. While industry attention focuses on launch systems and mission applications, the middleware powering satellite operations often remains neglected.

Legacy operating systems, designed for terrestrial environments, introduce excessive overhead, cybersecurity risks, and a lack of standardisation. Many satellite operators also rewrite software stacks from scratch, leading to long delays and reduced flexibility, as each mission must independently solve the same foundational challenges.

This is also true of the communication stack and cryptography in particular, which is paramount to mission security. While the CCSDS specifies protocols such as the Space Data Link Security Protocol (SDLS) [1] and its extended procedures (SDLS-EP) [2], it does not specify protocols for key establishment, or secure updates, or future-proof post-quantum cryptography, forcing users to roll their own solutions.

Furthermore, implementations of these protocols and their underlying cryptography are often kept private by their users. For the few available implementations such as NASA's `cryptolib`, vulnerabilities have been regularly discovered [1][2] despite significant testing and compliance efforts.

[1]https://securitybynature.fr/post/hacking-cryptolib/
[2]https://github.com/nasa/CryptoLib/security

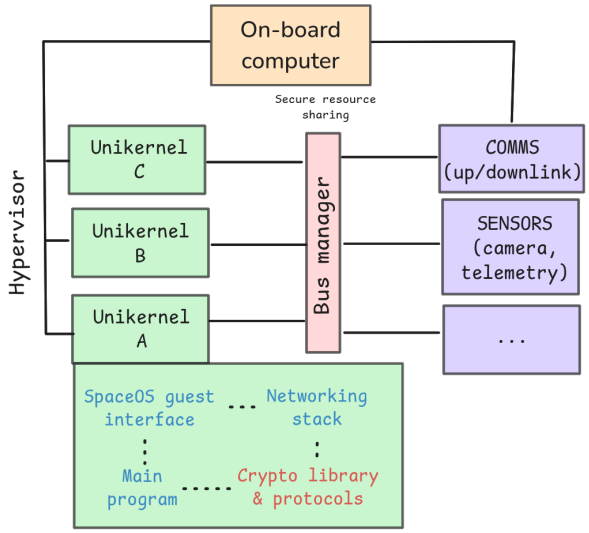

Fig. 1. Simplified architecture of a deployed SPACEOS system

These issues have led to increased attack surfaces in mission-critical components, forced satellite operators to allocate significant resources to patching vulnerabilities, and constrained mission flexibility by relying on outdated software stacks that are difficult to update once deployed.

At the same time, satellite operators are under increasing pressure to fully leverage their capacities, particularly in the areas of edge processing, but this is restricted by existing systems that lack the flexibility to integrate new technologies and applications developed by third-parties. Satellite operators lack the ability to dynamically install, update, and secure software in orbit from multiple sources. Satellites are purpose-built, and the mission is hardly adaptable to changing requirements. That being said, the introduction of new shared-payload capabilities introduces new cybersecurity risks (data leaks between payloads, malicious payloads affecting the satellite, etc.). Any solution addressing the market demand must carefully balance features and risks.

This paper proposes that solutions to both of these problems (reliable cryptography and robust operating systems) should go hand in hand, as the former is a key building block of the latter. Section II presents SPACEOS, a framework for secure deployment and execution of virtualized payloads. Sec-

tion III presents LIBCRUX, an open-source, formally verified implementation of a wide variety of cryptographic primitives and algorithms. Then, Section IV discusses the potential applications of a combination between the two, and focuses on one of them for a case study. Finally, Section V describes the next steps of this integration and concludes.

II. Secure Unikernels for Software-Defined Payloads

Unikernels [3] are specialized, single-address-space machine images constructed by combining application code with only the minimal set of operating system (OS) functionalities required to run that specific application. Unlike traditional operating systems that support a wide variety of applications and users, unikernels are tailored for specific tasks.

The benefits of this approach have to do with how the built unikernel is executed on actual hardware: either bare-metal (in which case the footprint on system resources is strictly minimal) or more commonly, virtualized by an hypervisor such as KVM, Xen or others.

That latter case makes unikernel well-suited for running potentially untrusted software payloads within a high-assurance context such as a cloud provider's datacenter (running their customer's software) or a satellite's on-board computer (OBC). Indeed, unikernels provide the benefits of both virtualization (strong memory and/or time partitioning) and containers (self-contained, lightweight and easily distributable).

In the context of earth observation, this enables payload developers to quickly iterate on software without the need for highly specialized tooling nor the need to rely on the satellite's operator to provide dependencies (libraries, runtime tools) required by the software. In turn, this ensures satellite operators that they can run third-party software without the risk of interference to other payloads or the mission itself.

There are multiple unikernel implementations available today. SPACEOS is a work-in-progress product that leverages MirageOS and Unikraft to run end-user applications:

MirageOS [3] is written in OCaml (a memory-safe programming language) and one of the first unikernel implementations. By design, it does not implement POSIX support in any way, instead relying on a set of its own Mirage interfaces and abstractions for communication between applications and host. On the other hand, platform and hardware support is extensive: KVM (solo5), Xen, Bhyve, muen, seL4 hypervisors, on various CPU architectures. Furthermore, the bytecode support for the OCaml compiler means an application written for unikernels may be executed on extremely limited targets such as MCUs or RTOS, as long as a C compiler is available.

Unikraft [4] is written in C and targeted towards cloud environments. It is based around the concept of small, modular libraries, each providing a part of the functionality commonly found in an operating system (e.g., memory allocation, scheduling, filesystem support, network stack, etc.). There is an important focus on POSIX compatibility, whereas a large subset of system calls is implemented, and an experimental ABI compatibility layer is provided.

Unikraft is harder to port to new platforms and architectures, and as such, the project mainly focuses on the KVM hypervisor (through qemu or firecracker), with Xen available as well. On the other hand Unikraft is known to be performant, through its efficient use of the hardware combined with the minimalism of its modular components' implementations.

Building on these two projects, SPACEOS is a work-in-progress toolchain to build a full operating system installable bare-metal on an OBC that bundles:

- a hypervisor (KVM, Xen, muen or seL4);
- the runtime necessary to execute unikernels (built with MirageOS or Unikraft);
- an implementation of core capabilities (networking stack, CCSDS implementations, on-board device access) exposed to the end applications, themselves implemented as unikernels (in a microkernel-inspired architecture, where the only trusted base is the hypervisor);
- a privileged "control plane" allowing dynamic addition, removal or updates of unikernels from the ground, as well as monitoring existing ones.

This architecture is depicted in a simplified manner in Figure 1, where multiple unikernels are deployed and accessing common resources. Core capabilities like networking and cryptography are exposed as readily available user-space modules calling the para-virtualization interface. Both for the core features exposed to the running applications (such as an SDLS implementation) and the control plane (in particular for secure application deployment), it is critical that SPACEOS embeds a reliable cryptography stack that has little risk of needing a replacement while in flight.

III. High-Assurance Post-Quantum Cryptography

LIBCRUX [5] is a formally verified cryptography library that provides a comprehensive set of algorithms, including those required by CCSDS [6] and needed by protocols like TLS [7]. The code for each algorithm is written in Rust and includes both portable implementations and those optimized for different platforms such as Intel AVX2, ARM Neon, and Arm Cortex-M4. LIBCRUX also contains Rust code generated [8] from the formally verified HACL⋆ project [9]. Importantly, LIBCRUX does not sacrifice performance: its code is as fast as, and sometimes faster than, other unverified implementations.

The Rust code in LIBCRUX can be used as-is, but can also be compiled to C for ease of integration into other software projects. As of writing, Rust code from LIBCRUX is used in Signal, and the compiled C code from LIBCRUX and HACL⋆ is used in multiple mainstream projects like Mozilla Firefox, OpenSSH, Linux, WireGuard, ARM mbed, Python, etc.

LIBCRUX supports both classic and post-quantum cryptography. It includes, in particular, Rust implementations of ML-KEM [10] and ML-DSA [11] that are easy to integrate into any post-quantum cryptographic protocol. For example, ML-KEM can be used to provide support for hybrid post-quantum ciphersuites of TLS, while ML-DSA can be used to sign secure software updates. Indeed, LIBCRUX is used as the main cryptographic provider for BERTIE [12], a formally verified

post-quantum TLS 1.3 implementation that is interoperable with Chrome, Firefox, and Cloudflare.

Developing High-assurance Cryptography in Rust. The root cause for many security and availability issues in legacy communication stacks can be traced to two inherent shortcomings. First, their implementations, like NASA's `cryptolib`, are written in unsafe languages such as C, making them vulnerable to a wide range of memory safety bugs that are easy to exploit. And secondly, they are not formally verified for correctness or side-channel protection, and hence provided limited guarantees against security bugs that are hard to find just with testing.

To address the first issue, the library authors use Rust to implement all of its cryptographic algorithms and protocols. Rust allows writing high-performance code without compromising memory safety, even in settings where a garbage collector is not feasible, and is heavily promoted as a replacement to C by governmental organizations [13], research institutions[3], and industry bodies[4].

To then increase assurance of cryptographic software to the level necessary for space applications, LIBCRUX's developers advocate the use of formal verification to ensure statically at compile time that the application can not crash, is functionally correct and is secure against side-channel attacks.

Verifying Cryptographic Software with HAX. The Rust code in LIBCRUX and BERTIE is formally verified using the HAX [14] toolchain and the F⋆ [15] proof assistant.

Figure 2 depicts the high-level workflow. The Rust developer can use the HAX toolchain to translate their source code to a formal model in the F⋆ language. The source code can be annotated with specifications in the form of pre- and post-conditions, loop invariants, etc. and these will get translated to assumptions and verification conditions in the F⋆ model.

The first property we can prove for the generated F⋆ model is runtime safety. Even though Rust is memory-safe, Rust code may still panic at runtime, or return an incorrect result, if (say) an array is indexed out of bounds or if an arithmetic operation over- or underflows. Both of these outcomes are dangerous for communication protcols and can lead to system malfunction. Using the F⋆ typechecker, we can formally prove the absence of panics in Rust, and more generally, that the source code meets all the pre-conditions of the (system and external) libraries it relies upon.

Once runtime safety has been shown, we can go a step further to show functional correctness of the implementation. In the LIBCRUX crypto library, the authors prove the correctness of the highly optimized implementations for various platforms by showing their functional equivalence to a high-level mathematical specification. In the BERTIE protocol implementation, they show the correctness of the message serialization and parsing code, to ensure that no network attacker may inject a message that may lead BERTIE to malfunction.

Finally, we can prove the relevant security properties for the cryptographic code. For LIBCRUX, we can prove that the code

[3]https://www.darpa.mil/program/translating-all-c-to-rust

[4]https://www.memorysafety.org/

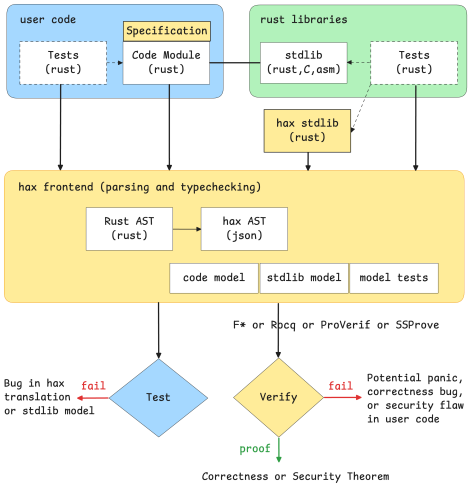

Fig. 2. Formal Verification with HAX

is *secret independent*, that is, its control flow does not depend on secret values, preventing a large class of side-channel attacks. Secret independence enforces a coding discipline (sometimes called "constant-time programming") that forbids operations like branching on secrets or using secrets as indexes to arrays which are known to result in exploitable side-channel vulnerabilities. Note that this doesn't remove the need to consider the low-level hardware and firmware behaviour when assessing the security of a real deployment environment.

For BERTIE, we can verify the security of the core TLS protocol code by using HAX to translating this code into an input for the ProVerif tool [16], which is then able to analyze the prove the implementation is not vulnerable to various classes of symbolic protocol attacks [12].

IV. Integrating LIBCRUX in SpaceOS

The main contribution of this paper is the integration of LIBCRUX as a cryptographic provider for SpaceOS, which enables the use of high-assurance post-quantum cryptography in satellite applications. We make both LIBCRUX and BERTIE available both to the communication stack within SpaceOS and to applications running within the unikernels. We identify several potential use cases, and have implemented some applications as proof-of-concept demonstrations:

- **Secure Software Updates:** We show how to implement secure updates for Satellite software using post-quantum secure signatures (detailed in the section below.)
- **Secure Channels:** By making BERTIE available to unikernels, we provide secure channels for ground-space and space-space communications. For example, a ground operator can use a standard Chrome or Firefox browser to connect to a satellite application via post-quantum TLS 1.3, something that is only made possible by our integration. Similarly, two satellites running SpaceOS can confidently communicate with each other using formally verified implementation of TLS 1.3.

- **Key Establishment for SDLS-EP:** While the SDLS protocol [1] specifies symmetric-key based encryption and authentication constructions, neither SDLS nor its extensions [2] specify how these symmetric keys may be established or distributed. This is in contrast to terrestrial protocols that commonly use public-key protocols based on elliptic curves or (more recently) post-quantum cryptography to establish symmetric keys. Using LIBCRUX, users can build and deploy any standard or customized key exchange protocol. In particular, they can even use BERTIE as a key establishment component to set up symmetric keys to bootstrap subsequent SDLS communications.
- **Authenticated Channels for Quantum Key Distribution**: A number of projects are experimenting with the use of satellites for Quantum Key Distribution (QKD), including the QKDSat project of the European Space Agency [5]. The security of QKD protocols relies on an underlying post-quantum secure authenticated channel, and no such protocol has been deployed to date. However, using ML-DSA and ML-KEM, a user can design, build, and deploy quantum-resistant secure channels from ground to space. In particular, a fully post-quantum extension of BERTIE could also be used as an authenticated channel for a QKD application running on SPACEOS.

Proof of Concept: Secure software updates using signed unikernels. As a first proof of concept for integration between unikernels and reliable cryptography, we ported a part of LIBCRUX to add quantum-proof secure updates to SPACEOS. More concretely, the SPACEOS unikernel management component (which handles deployment, monitoring and update of the on-board unikernels) was extended so the operator could specify a public key to signify the expectation that every new unikernel deployed on-board should be signed with its corresponding private key.

While the notion of secure updates is not novel, the interesting part was using the ML-DSA [11] implementation of LIBCRUX to that end. This implementation being written in Rust, we exported direct bindings to OCaml (used to write SPACEOS) [6] , without the need for a C transpiling unnecessary and therefore reducing the amount of intermediate steps between the formally verified code and the actual runtime. In this mode, every unikernel was expected to be prefixed with a signature of its actual body, and this signature was verified at the time of deployment.

The bindings to OCaml were written to mimic the signature/verification interface of `mirage-crypto`[7], to enable any OCaml application already using this interface to simply swap the implementation to LIBCRUX's ML-DSA implementation in the future.

[5]https://www.esa.int/Applications/Connectivity_and_Secure_Communications/Secure_communication_via_quantum_cryptography

[6]While SPACEOS remains closed-source, the bindings were made public: https://github.com/parsimoni-labs/ocaml-libcrux

[7]https://github.com/mirage/mirage-crypto

V. ONGOING AND FUTURE WORK

The proof of concept described in the section above is minimal, but the smoothness of its execution has encouraged the authors to explore deeper integrations. In particular, we are working on making LIBCRUX's ML-DSA implementation available to the (MirageOS) unikernel themselves.

Indeed, MirageOS unikernels are just specific OCaml applications that are tightly coupled with both the low-level MirageOS interfaces (to access block devices, network interfaces, etc.) and the high-level ones (e.g. networking stack, **cryptography**) which build on them. As a result of this, the way all MirageOS unikernels perform even high-level operations (such as cryptographic operations) is mostly standardized. This enables us to seamlessly swap `mirage-crypto` provided implementations by LIBCRUX ones without changes to the user code. While a signature implementation is interesting, expanding the subset of LIBCRUX features exposed to MirageOS applications would be ideal: its TLS implementation in particular would be of value.

Beyond that, the authors are exploring applications listed in Section IV and in particular integration with a built-in SPACEOS SDLS-EP implementation, as a step towards all space software payloads benefiting from built-in, formally verified, quantum-proof communication protocols with little friction for the developers.

REFERENCES

[1] The Consultative Committee for Space Data Systems (CCSDS), "Space Data Link Security Protocol (Blue Book)," CCSDS Secretariat, National Aeronautics and Space Administration, Tech. Rep. CCSDS 355.0-B-2, July 2022. [Online]. Available: https://ccsds.org/Pubs/355x0b2.pdf

[2] ——, "Space Data Link Security Protocol–Extended Procedures–Summary of Concept and Rationale (Blue Book)," CCSDS Secretariat, National Aeronautics and Space Administration, Tech. Rep. CCSDS 350.11-G-1, July 2024. [Online]. Available: https://ccsds.org/Pubs/355x0b2.pdf

[3] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: library operating systems for the cloud," *SIGARCH Comput. Archit. News*, vol. 41, no. 1, p. 461–472, Mar. 2013. [Online]. Available: https://doi.org/10.1145/2490301.2451167

[4] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, c. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, and F. Huici, "Unikraft: fast, specialized unikernels the easy way," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 376–394. [Online]. Available: https://doi.org/10.1145/3447786.3456248

[5] Cryspen, "libcrux - The formally verified crypto library for Rust." [Online]. Available: https://github.com/cryspen/libcrux

[6] The Consultative Committee for Space Data Systems (CCSDS), "CCSDS Cryptographic Algorithms (Green Book)," CCSDS Secretariat, National Aeronautics and Space Administration, Tech. Rep. CCSDS 350.9-G-2, July 2023. [Online]. Available: https://ccsds.org/Pubs/350x9g2.pdf

[7] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018.

[8] A. Fromherz and J. Protzenko, "Compiling C to Safe Rust, Formalized," 2024. [Online]. Available: https://arxiv.org/abs/2412.15042

[9] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A verified modern cryptographic library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1789–1806.

[10] National Institute of Standards and Technology (NIST), "Federal Information Processing Standard (FIPS) 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard," U.S. Department of Commerce, Tech. Rep. FIPS 203, August 2024, effective on August 14, 2024. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.203

[11] ——, "Federal Information Processing Standard (FIPS) 204: Module-Lattice-Based Digital Signature Standard," U.S. Department of Commerce, Tech. Rep. FIPS 204, August 2024, effective on August 14, 2024. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.204

[12] Cryspen, "Bertie TLS 1.3 Implementation," https://github.com/cryspen/bertie, 2022, accessed: 2025-06-28.

[13] "Back to the building blocks: a path towards secure and measurable software," https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf, White House Office of the National Cyber Director, Feb 2024.

[14] K. Bhargavan, M. Buyse, L. Franceschino, L. L. Hansen, F. Kiefer, J. Schneider-Bensch, and B. Spitters, "hax: Verifying security-critical rust software using multiple provers," in *Verified Software. Theories, Tools and Experiments - 16th International Conference, VSTTE 2024, Prague, Czech Republic, October 14-15, 2024, Revised Selected Papers*, ser. Lecture Notes in Computer Science, J. Protzenko and A. Raad, Eds., vol. 15525. Springer, 2024, pp. 96–119. [Online]. Available: https://doi.org/10.1007/978-3-031-86695-1_7

[15] N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Z. Béguelin, "Dependent types and multi-monadic effects in F*," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, R. Bodík and R. Majumdar, Eds. ACM, 2016, pp. 256–270.

[16] B. Blanchet, "Automatic verification of security protocols in the symbolic model: The verifier proverif," in *FOSAD*, ser. Lecture Notes in Computer Science, vol. 8604. Springer, 2013, pp. 54–87.