

Isle: Network-level Isolation for Low-power Embedded Applications

Marshall Clyburn
marshallc@virginia.edu
University of Virginia
Charlottesville, Virginia, USA

Brad Campbell
bradjc@virginia.edu
University of Virginia
Charlottesville, Virginia, USA

Abstract

The transition toward a unified low-power networking protocol enables seamless interconnectivity in smart and industrial IoT ecosystems. However, this inherent interconnectedness exposes embedded devices to lateral threats, where buggy or malicious applications can impact the resources, stability, and security of unrelated devices through undesirable network interaction. To resolve this, we introduce Isle, an OS-level facility that provides network-level isolation for individual low-power embedded applications. Isle works by subsuming traditionally application-layer security logic into the embedded operating system, additionally using it to authenticate packet sources and to enforce logical communication boundaries. We build and deploy Isle to a real testbed network of low-power devices and evaluate it. Our evaluations find that Isle effectively isolates applications at the network level and imposes sub-millisecond additional latency overhead on packet processing; the isolation also improves the energy usage of devices by shielding them from unrelated traffic, reducing energy usage of an application by 23%.

CCS Concepts

• **Computer systems organization** → **Embedded and cyber-physical systems**; *Embedded software*; **Sensor networks**; • **Software and its engineering** → **Operating systems**; • **Security and privacy** → **Security protocols**; *Domain-specific security and privacy architectures*; *Access control*.

Keywords

embedded systems, sensor networks, operating systems, network isolation, shared sensor network, virtual sensor network

1 Introduction

The need to strengthen connectivity and reduce deployment costs of embedded devices is driving the IoT’s transition from a fragmented field of low-power networking protocols to a convergence on unified mesh networking interconnecting all devices in the smart ecosystem. The Thread protocol [34] has emerged as the common, low-power mesh networking technology to connect devices in smart homes [3, 6], smart buildings [5, 7], and Industrial IoT [4]. In addition to removing the burden of managing multiple, mutually incompatible wireless protocols, its flexibility as a mesh networking technology removes the complexity of configuring and maintaining central hubs for connectivity. Thusly, it simplifies the development of new applications and the integration of new devices. Such a networking technology moves the current state of the IoT toward its vision of enabling powerful, connected applications in automation, health, manufacturing, and more.

At the same time though, this sprawling interconnectedness increases the opportunity for problematic interaction between applications. In reality, applications are buggy, not robust to unexpected conditions, misconfigured, or employ suspect or even outright malicious behavior. Whereas the prior heterogeneity in network protocols provided strong, natural segmentation between these devices and applications, a unified mesh network lacks those boundaries. Formerly sequestered applications are able to interact with each other and produce undesirable results. For example, an occupancy detection sensor may broadcast messages about room occupancy changes which allows lighting in the room to automatically turn on and off but also triggers a response from a smart lock, wasting its resources.

The ability to constrain an individual application’s network interaction is essential to maintaining the same safety, security, and reliability of running these applications in disjoint networks. In single-user infrastructure, like the smart home, the user should be able to constrain less trusted applications in the network, conserve the resources of energy-constrained devices, preserve application performance, and add a layer of protection to prevent lateral interactions with suspect applications. The same also applies to multi-tenant infrastructure, in which multiple stakeholders will deploy applications: such as a university building with building management, IT employees, workers, and researchers. Additionally, for multi-tenant infrastructure, the network operator should be able to guarantee to tenants that their applications will only ever see their own traffic, fulfilling information security requirements [15, 21], or that the network will maintain some quality of service.

Existing computing technologies for security or traffic management are either inapplicable to constrained IoT mesh networks or are inadequate for providing isolation. Deploying firewall devices into the mesh is infeasible; the network structure is dynamic, making them unreliable and their circumvention inevitable once the network topology changes. VLANs, even in conventional computer networks, require specialized hardware and, as with a firewall device deployed in the mesh, would fail to maintain isolation due to the network’s amorphous structure. Using TLS may provide connection authenticity, but it does not entirely prevent network interaction; an energy- or resource-constrained device would still suffer resource consumption from connection attempts. And link-layer encryption and authentication only prevents access to the network and does not impede devices and applications that the user has allowed into the network.

The low-power mesh network of embedded systems also presents its own characteristic challenges. The network features small maximum transmission units (MTUs), leaving little capacity for additional overhead for data identifying a frame’s source or purpose.

Receiving and transmitting larger frames also reduces operational lifetime for energy-constrained devices. And the devices themselves have limited computing capability. Performing additional processing to realize managed networking will increase processing time.

However, we observe that the components and technologies key to achieving network-level isolation are already present, and by arranging them fittingly, we can to attain low-overhead network-level isolation for applications in the constrained network. Networked applications already utilize cryptography for communication security. That same layer of security can also extend to provide network isolation for applications. Controlling and varying cryptographic parameters (e.g., secrets, keys, IVs) leaves communication intelligible to only a single recipient or group of recipients. It can simultaneously provide an authenticity guarantee about the source of data. If application-layer security becomes manageable by the network owner, then it would empower them to selectively determine which applications on the network can communicate with each other while maintaining the security of data in flight.

To attain low-overhead, network-level isolation for low-power embedded applications, we leverage this core idea and design Isle. Isle is an OS facility that monitors and processes application network traffic to control what data an application can send and receive. Operating within the end device's embedded operating system, it requires no new hardware be deployed in the network. Instead, Isle subsumes traditionally application-layer security as a responsibility into the embedded OS, and, while maintaining communication security for applications, simultaneously leverages its responsibility for cryptographic functions to restrict applications' network communications to defined groups. By re-using application-layer cryptography to additionally enforce network-level isolation, Isle avoids the overhead of an entirely separate layer.

We implement Isle within an embedded OS to demonstrate its feasibility and evaluate its utility and overheads. We deploy Isle in a testbed of low-power embedded devices that form a Thread mesh network and evaluate the performance of Isle as well as the performance of applications running under Isle's isolation in several evaluations in order to understand the applicability of the system. The results show that Isle works to enforce network-level isolation on applications, shielding an application from receiving a misdirected request reduces device energy usage by 23%. And when enabled, Isle's operation adds less than a millisecond of processing time overhead compared to an application with no application-layer security.

2 Related Work

Essentially, Isle is a work in network-level WSN (wireless sensor network) virtualization: repurposing existing network infrastructure to create logical links between a subset of nodes. However, whereas WSN virtualization constructs overlay networks to provide connectivity between indirectly connected nodes, Isle expands beyond existing work, exploring how to strictly isolate applications in the network with sensitivity to packet and latency overhead.

The topic of WSN virtualization is broad. Khan et al. provide a survey [24] detailing works employing virtualization techniques. Many early works, like Jayasumana et al.'s [23] and Melete [39], propose VSNs (virtual sensor networks) by dedicating a portion

of or the entire network infrastructure to a single application at a time, effectively turning the network into a time-sliced resource. More recent works still explore this time-slicing approach but with focus at the MAC layer [17, 38]. Sensomax [20] deploys a selection of one or more applications to a defined cluster of nodes in a VSN. These forms of WSN virtualization provide a restrictive form of isolation but have implications about when and where applications can run. Isle does not impose simultaneous scheduling or limited transmission window restrictions on the application.

More recent works abstract the network infrastructure from the application layer and enable multiple, concurrent applications to share the network. Sensshare [19, 25] enables multiple applications to run concurrently on an IEEE 802.15.4-based network, achieving isolation between applications by adding a 6-byte routing header to each frame. The method discerns one application's traffic from another but is not suitable for a network with untrustworthy applications. IoT-VN [22] proposes secured virtual networks containing only relevant devices using a simple tunneling protocol but lacking security. In contrast to these works, Isle works for a network of concurrently running applications and provides security as part of its isolation capability.

Software-defined networking (SDN) and network function virtualization (NFV) for the IoT propose solutions for managing traffic of networked embedded devices [9]. These approaches either only control traffic at the network boundary [27] or require a mediating gateway between devices [13, 18]. Isle avoids the deployment of additional hardware and uses the embedded OS to effectively implement a per-application firewall, a generally related idea. Isle also does not centralize communication through a gateway, retaining the routing efficiency of the mesh network.

There are also many existing computing technologies and standards that offer isolation. The Bluetooth Mesh [11] standard defines "Models" which use "AppKeys" to secure communication to a subset of nodes. The Matter [16] standard is an application-layer security protocol for communication between IoT devices. Both offer secure group-level communication but have strong application design constraints that applications must fit to use them. Isle provides secure one-to-one and one-to-many communication that is compatible with any application using TCP/IP.

3 System Design

Isolating an application at the network level means dictating which applications running on the network can exchange traffic with each other. If an application is isolated from another application, the two cannot exchange traffic. Principally, Isle achieves this by tracking the logical groups the application running on the device is a member of and discerning which group a packet originates from or is destined for. If the application attempts to send to a group that it is not a member of, Isle does not allow the transmission. And if the device receives a packet from a group that the application is not a member of, Isle does not pass the payload to the application. Applications cannot communicate outside of their groups. Isle leverages the privilege and access of the embedded OS underlying the application to determine whether to pass or drop packets as they arrive or depart, like a specialized firewall.

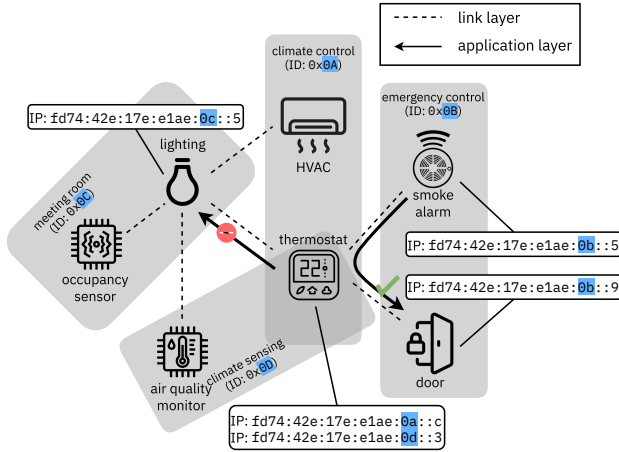


Figure 1: An overview of Isle-enabled devices assigned to isolating realms. Isle enables the network owner to logically group application instances together to constrain the application instances they can communicate with despite the network’s topology. In this network, the thermostat cannot interact with the lighting at the application layer despite their connection at the link layer; but the smoke alarm can operate the door in case of an emergency even though they are not directly connected.

In addition to creating network-level isolation, key secondary goals influence its design:

- *Minimize frame size overhead.* Low-power networks have low maximum transmission units (127 bytes for Thread). Maximizing application payload space reduces fragmentation, improving network performance and reliability.
- *Enable cryptographic agility.* Static security implementations eventually fail to meet evolving best practices and requirements. Enabling seamless adoption of new cryptographic implementations ensures that the network-level isolation remains secure and robust over time.
- *Retain compatibility with CoAP-based protocols.* Smart and IIoT environments heavily use CoAP and CoAP-based protocols [2, 10] which offer features that improve communication over constrained networks (e.g., proxying, caching). Avoiding altering packets in a way that makes these techniques impossible will keep the network efficient.

The isolation that Isle offers targets applications that interact with other devices within the network. For example, building automation systems monitoring and controlling safety and security, occupancy sensors communicating with lighting and room equipment, and climate and control sensors and equipment. Logically segmenting the network these systems share contributes to their security and stability while helping conserve constrained devices’ resources.

3.1 Achieving network-level isolation

Network-level isolation must work regardless of unpredictable, topological shifts in the mesh network caused by nodes joining,

leaving, or switching between leaf and router roles. But enforcing a structure on the topology and inserting dedicated hardware to implement isolation, akin to firewalls or L3 switches, requires the deployment and configuration of additional devices and limits the mesh network’s fundamental strength in flexibility. In lieu of specialized hardware or nodes within the network, we design Isle to run as system-level software directly on the device running the application. Then, as part of system software, Isle can monitor the application’s network traffic and prevent unapproved packets from leaving or reaching the application based on its source or destination, regardless of the exact topology of the network. Acting within the system, Isle can stop the first packet of an application-layer interaction or handshake from reaching the application, conserving resources that would have gone to exchanging further packets or processing a request.

In order for Isle to control which payloads the application can send or receive, we position it directly into the control flow of packet processing in the network stack (as another “layer”). There is flexibility in where it works in the network stack, but having it work in the upper part of the network stack (layer 3 to 5 in the five-layer model) provides more advantages than placing it lower in the network stack. The data that it is concerned with isolating and the data most relevant to Isle’s operation (application data and payload source and destination) are in the upper part of the network stack. Working closer to the data of concern keeps its design simpler. This also keeps Isle more portable to other underlying networking technologies.

Determining whether to allow a packet to pass to or from the application is partly achievable by inspecting source and destination addresses, but it does not prevent a packet with a spoofed address or a replay packet from reaching the application and breaking the isolation. Completing the isolation additionally requires authenticating packet sources. Adding authenticating cryptography to verify senders attains the goal of isolation but comes at the cost of additional overhead for each packet. The system must expend processing time for calculating message authentication codes (MACs), and the inclusion of the resulting tag decreases space available in each packet for application data. This runs counter to the goal of minimizing frame size overhead. It also requires transmitting and receiving more bytes for each packet, impacting operational time for energy-constrained devices.

Isle’s exact positioning within the networking stack becomes critical to ameliorating this overhead. Applications are likely to already use cryptography to secure their communications. By placing Isle adjacent to the application layer, it can take advantage of its positioning to conserve processing and data overheads. Instead of performing two separate cryptographic operations (one within the application and one for Isle’s isolation layer), a single use of cryptography can meet the needs of both the application and the isolation capability. The application can have the assurances of confidentiality, integrity, and authenticity; and the same verification of authenticity can provide assurance to Isle that the payload came from its stated source. Therefore, **Isle’s design proposes that the application delegate responsibility for communication security to the underlying system.** This then allows Isle to additionally use this security to also provide network-level isolation and avoid the overhead of an entirely separate layer.

3.1.1 CoAP compatibility with Group OSCORE. Taking responsibility of the application’s communication security allows the system the choice of how to secure communication. However, many existing applications on constrained networks make heavy use of CoAP [31] and CoAP-based protocols, like LwM2M [28]. These protocols’ designs aim to keep communication on the network efficient with features like proxying and caching which involve intermediary nodes that are otherwise unrelated to a device. Encrypting an application’s entire payload provides complete confidentiality from such intermediaries but decreases the efficiency of the network and breaks features such as proxying and load balancing.

To preserve the operational efficiency of the network and devices, Isle leverages Group OSCORE [30, 36] to secure payloads from applications. Group OSCORE provides confidentiality, integrity, and authentication and also preserves metadata that is critical for efficient, low-power network operation. It also enables both one-to-one and one-to-many modes of communication. Isle handles transforming packets to and from Group OSCORE. Group OSCORE can also carry payloads that are not CoAP messages.

3.2 Isolating applications with realms

In Isle, a **realm** defines the bounds of communication for applications. A realm contains one or more **application instances**, the installations of applications on devices. Application instances that are members of a particular realm can only communicate with members in that same realm, and non-member application instances cannot communicate with member application instances, even if the two application instances are the same application. This provides the granularity to, for example, contain the network activity of room occupancy sensing devices to just other devices within their own rooms (such that they can only power on lighting or appliances in that same room).

To successfully isolate application instances’ traffic within the realm, Isle must be able to discern the realm that traffic belongs to. One straightforward way of determining which realm a packet belongs to is to maintain a collection of addresses of all members of the realm on the device and consult that data when processing incoming and outgoing packets. However, this is infeasible for devices which feature limited storage and memory, and it does not scale well to large networks.

Instead, Isle extends some control over IPv6 address selection for application instances in the realm to make filtering more efficient. IPv6 networks usually combine a 64-bit network prefix with a single, large 64-bit address space, and networked devices typically randomly choose an address from that space. Isle uses the upper 16 bits of that randomized address space to indicate the realm a packet is in. This effectively creates 2^{16} smaller /80 networks, but still leaves 2^{48} addresses in each realm available for use. Importantly, this means Isle imposes an addressing requirement on networked applications. Isle must be able to determine at least a portion of the address an application transmits data with. For example, application instances on a network with the prefix fd74:42e:17e:e1ae/64 that are members of the realm with realm ID 0xAE12 must have an address matching fd74:42e:17e:e1ae:ae12/80. In exchange, Isle can filter many packets by inspecting 16 bits. There is no restriction

Parameter	Description
realm ID	local network-unique 16-bit identifier
master secret	Group OSCORE key derivation secret
master salt	Group OSCORE key derivation salt
sender sequence no.	message counter for IV derivation

Table 1: Realm context parameters. The OS maintains contexts for all realms that the application is a member of.

on the lower 48 bits of the address. Figure 1 provides an overview of how realms function.

Isle uses Group OSCORE (Section 3.1.1), rather than an entirely new cryptographic process, to protect and authenticate communications within the realm. Group OSCORE supports both the one-to-one and one-to-many modes of communication, mapping well to IPv6’s unicast and multicast methods of communication. We combine Group OSCORE’s common context together with a realm’s parameters to form the realm’s context (Table 1). Each device with an application instance participating in the realm must keep track of the realm context. And some applications may require the ability to communicate in multiple realms. To enable reception of packets from more than one realm, Isle assigns multiple addresses to the application instance. To enable the transmission of packets in more than one realm, Isle uses the address that matches the realm identifier as the source address for an outgoing packet.

With Isle, the network owner gains the ability to manage application network communication security and to control the bounds of those applications’ communication. This requires managing each realm’s shared context deployed on Isle-enabled devices. Because Isle is integrated with the embedded OS and not the application, adding or removing the realm context from a device hosting an application instance is sufficient to prevent the application instance from communicating in the realm. Isle would then no longer allow the realm’s traffic to or from the application instance. The best practice, however, would be to change the master secret and master salt of the realm which Group OSCORE uses to secure communication.

3.3 Network stack integration

Isle’s integration into the network stack of the networked end device is essential to its function. Adding Isle’s isolation “layer” into the network stack enables seamless processing of payloads as the application transmits them and the device receives them. There is no work in locating the application layer data compared to operating closer to the radio and less data to copy between buffers, yielding reduced processing time overhead. This design choice is what enables Isle to take on the responsibility of application-layer security. Functioning just under the application layer also supports its secondary goal of enabling cryptographic agility. The embedded OS can expose a common interface for configuring or replacing cryptographic implementations, an essential capability for long-term security [12]. With Isle, application vendors are no longer responsible for ensuring their applications meet best practices in communication security. Isle can also handle making use of the microcontroller’s available accelerators to run cryptographic operations, providing quick and efficient security without the application needing to explicitly make use of the hardware.

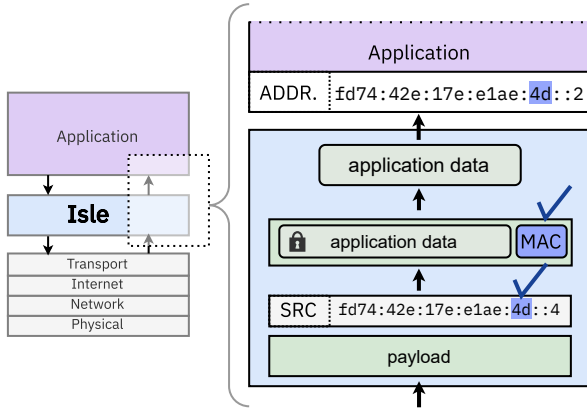


Figure 2: Isle’s integration into the end device’s network stack. Isle logically works as an additional layer in the stack to both secure an application’s communication to other applications on the network and enforce restrictions on where the application can send and receive communication to and from. Its network addressing scheme and cryptographic authentication (highlighted and colored in blue) ensure only other application instances in the same realm can communicate with the application. In this depiction, the arriving packet’s source address and authentication tag match, allowing the payload to reach the application layer.

Within the network stack, Isle is responsible for enforcing realm isolation and transforming packets between application payloads and Group OSCORE messages. Isle processes every packet moving to and from the application. Because this processing occurs just under the application layer, Isle does not interfere with packets for network management (e.g., link establishment, neighbor discovery) and does not incur processing overhead on such packets. This also means that Isle is agnostic to physical and network layer and can easily work on a different medium.

When an application invokes the call to transmit a payload, Isle accepts the payload for processing. First, it determines whether to allow the application to send the payload by checking the destination address, which contains the realm ID. If Isle does not have the cryptographic credentials that correspond to that realm ID, then the application is not a member of that realm, and Isle drops the payload. For valid realm destinations, Isle uses the associated cryptographic credentials to secure the application payload as a Group OSCORE message, which includes providing source authentication, and passes the transformed payload to the next layer.

The reverse process occurs when the device receives a packet bound for the application. Upon receiving the packet, Isle first checks the source address to ensure the sender belongs to a realm the application is a member of. If the 16-bit realm ID encoded in the source address does not match a realm ID for the application, Isle drops the packet, and the application does not interact with the packet. If Isle finds a set of cryptographic credentials associated with the realm ID, Isle verifies the authenticity of the packet, decrypts the payload, and passes it upward to the application. Figure 2 depicts

Isle integrated into the system’s network stack and processing an inbound packet.

We defer to the Group OSCORE specification [36] to avoid redundancy in details of its operation, but in using Group OSCORE, Isle performs some implementation-specific behavior which the specification intentionally leaves open to maintain adaptability to different use cases. The group ID, which distinguishes Group OSCORE contexts among others, comes from and is the same as the Isle realm ID. The sender ID, which must be unique for every member of the group, comes from the 7 least significant bytes of the application instance’s address.

3.4 Application requirements

The design of Isle expects that applications fully delegate communication security to Isle instead of implementing their own. This asks the application developer to omit the usage of any secure data communication facilities within the application code. Existing applications that implement their own secure communication would need to be rebuilt by the vendor to fulfill this expectation. But while Isle *expects* the application to delegate communication security to it, application code does not *have* to change to work under Isle’s isolation, provided that the applications it communicates with do the same.

Fully handing off communication security to Isle reduces packet data and processing overhead compared to an application deployed both utilizing its own communication security *and* running under Isle’s isolation. An application still implementing its own communication security and running under Isle’s isolation has two, immediately successive layers of security applied to its payloads. Each layer of security imposes its own packet data and processing overheads, resulting in less space available for application data and longer processing times per packet. Applications that transmit oversized payloads will exceed the frame MTU and incur packet fragmentation, negatively impacting performance and reliability.

Isle’s design aims to keep the application oblivious to its operation and to place control into the network operator’s hand. In so doing, Isle minimizes code changes to the application and avoids adding requirements for platform-specific code, helping to keep applications portable and their code maintainable. The application neither has the ability to disable Isle nor alter its operation.

4 Implementation

We implement Isle using both the Tock embedded OS [26] and the OpenThread [1] network stack. Its implementation demonstrates the feasibility of Isle’s design in existing embedded software. Tock provides strong separation between the OS and the application. The code for each is completely separate and independently deployable from each other. Tock OS is the system level software that we integrate Isle into to provide it the privilege and access necessary to enforce isolation. We implement Isle as a driver that runs in the Tock kernel. OpenThread provides an implementation of the Thread protocol a device can use to communicate on a Thread network. We modify OpenThread to add Isle into the network stack to process payloads as they move to and from the application. OpenThread is not integrated directly into Tock. Instead, we compile and deploy the Isle-augmented OpenThread library with the application. Even so,

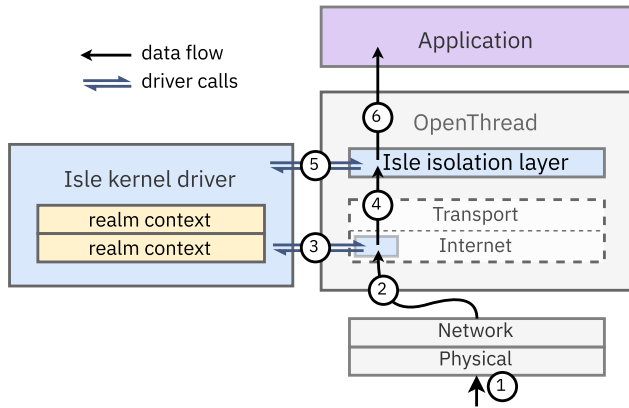


Figure 3: Implementation of Isle showing packet reception for an isolated application instance. The Isle isolation layer in the network stack and the kernel driver work in conjunction to enforce network-level isolation. After the Internet layer parses the packet, the source address enables Isle to perform early filtering of the packet to check its origin (3). The network isolation layer and kernel driver then decrypt and verify its origin (5) before passing the payload to the application (6).

the application still only makes use of OpenThread’s API to network, meaning that the application is compatible with an implementation that fully separates the application from the network stack.

4.1 Network stack changes

We modify OpenThread to process payloads that the application sends and payloads the device receives that are bound for the application. Isle’s operation in the network stack maps application payloads to and from Group OSCORE messages and checks that packets bound for the application have a valid realm ID in the address. Isle’s network stack component works in conjunction with the kernel driver to encrypt, decrypt, authenticate, and verify payloads. Though this portion of Isle runs in userspace in this implementation, the system retains the final decision to drop or continue processing the payload.

The Isle-augmented OpenThread stack supports transforming payloads between CoAP and Group OSCORE, but this functionality must be enabled on a per-application basis. There are several characteristics of CoAP that would help deduce whether a payload is likely a CoAP message (version field, code class and detail, payload marker, etc.), but these markers cannot certainly identify a payload as a CoAP message. This makes a parsing operation providing automatic detection susceptible to false positives. A false positive could then result in bad CoAP headers and options metadata which would affect correct operation of devices making use of such data. Whether to apply CoAP-compatible processing is controlled by an additional parameter in the local realm context.

4.2 Isle kernel driver

A driver within the Tock OS implements most of Isle’s logic. This driver maintains contexts for all realms an application instance is a member of and uses them to encrypt, decrypt, authenticate, and verify payloads as they pass through Isle. The driver exposes a syscall interface that the Isle-augmented OpenThread network stack uses to exchange data with the driver. As Isle supports not only the security of the application but also its isolation, it exposes no API to the application and cannot be disabled by the application.

The authenticated encryption algorithm the driver uses is swappable. The implementation abides by an AEADProvider interface that defines `encrypt()` and `decrypt()` functions suitable for both synchronous, software-based implementations and asynchronous, hardware-accelerated operations. Therefore, an OS update can transparently change the application’s communication security to migrate from weakened encryption algorithms or to adhere to new security practices.

The Isle OS driver is also flexible about how it can receive realm contexts for application instances. At startup, the driver has no realm contexts to process packets with but is initialized with an `IsleConfigurationProvider` which provides it with realm contexts. How it provides the driver with realm contexts is up to the implementer. This allows the driver to, for example, obtain realm contexts from a location in programmable flash storage or fetch them dynamically over the network. Figure 3 provides an overview of the implementation and packet reception.

4.3 Packet processing

When an application transmits a payload, the payload passes to the Isle network stack layer. In the network stack, Isle prepares the payload for encryption with Group OSCORE. When not using CoAP-compatible processing, Isle uses the entire payload without further processing. For CoAP messages, Isle forms the plaintext consisting of the header the payload, and the additional authenticated data according to the OSCORE standard. The network layer then hands the buffer to the kernel driver which uses the realm context it maintains on behalf of the application to encrypt the payload. If the payload’s destination is outside of the application’s realms, the driver will fail to find a realm context return an error, preventing the transmission of the packet. The kernel driver performs authenticated encryption on the payload and can accelerate cryptography using hardware available on the device or use a software-based implementation. Once the operation is complete, the kernel driver notifies the Isle network stack layer to continue processing the payload. The Isle network stack layer builds the final CoAP payload using the ciphertext and then invokes the transport layer to continue creating the final packet.

Payload reception is largely the transmission process reversed. However, for reception, Isle verifies the packet source once the network stack completes parsing the Internet layer. We modify the network stack to verify the source address with the kernel driver and drop the packet early if it does not originate from within the realm. Otherwise, once the payload passes upward out of the transport layer, the Isle network layer forms the additional authenticated data and requests the Isle kernel driver to decrypt. After verifying the packet’s authenticity, the kernel driver passes

the decrypted payload back to the network stack, which delivers it to the application.

5 Evaluation

Using our implementation of Isle, we performed a series of evaluations to validate its effectiveness, to understand how it affects the performance of networked applications, and to ascertain its packet data, memory, and code size overheads. Our testbed for evaluating Isle consists of eight nRF52840 DK [8] development boards. The nRF52840 microcontroller contains an ARM Cortex-M4 MCU featuring 1 MB of flash and 256 KB of SRAM. Its specification is representative of the low-power embedded hardware that we design Isle to run on.

The nRF52840 DK development boards form a mesh network we use to run our experiments. Three of the nRF52840 DK boards act as routers and connect the other five boards that act as end devices. The router devices run the Zephyr embedded OS (v3.6.99, nRF SDK v2.7.0), and the end devices run the modified version of Tock and OpenThread (based on v2.2 rev. dc7300be6, rev. a3e804101, respectively). Using Zephyr for the router nodes makes configuration for experiment setup easier and provides a stable, well-supported Thread implementation to support our experimental changes in the end devices. Each evaluation provides further experiment-specific details on the testbed setup.

We use a Raspberry Pi 2 [29] and a TI INA219 [33] to collect energy measurements, and we use an oscilloscope to collect precise timing measurements. The Raspberry Pi 2 also provides control over the start and stop timings of experiments, providing resets and GPIO inputs to signal to the devices under test.

We compare Isle with other application security configurations. The **no protection** configuration runs applications without any security protecting their communications atop Tock OS v2.2 unmodified. The **application-layer security** configuration runs applications with built-in DTLS support atop Tock OS v2.2. This configuration has two variants: application-layer security implementing cryptography in software and application-layer security utilizing the on-device cryptography accelerator. The **Isle** configuration runs applications with no in-built security, using Isle’s OS facility to provide both security and isolation. Isle and the application-layer security configurations use AES-128-CCM to secure communications. It is a common encryption algorithm used in communication security, and is often accelerated with dedicated peripherals in microcontrollers, making it a representative cryptographic choice.

5.1 Isolation effectiveness

In our first set of experiments, we establish the efficacy of Isle’s isolation by comparing the energy usage and active time across scenarios both with and without Isle.

5.1.1 Problematic scenarios. We first demonstrate Isle’s effectiveness and its ability to conserve the resources of applications and devices in different scenarios. Applications sharing the network can receive traffic from unrelated sources and they will respond in different ways. This evaluation highlights these possible reactions to unexpected traffic, their effect on the recipient, and how Isle ameliorates these scenarios.

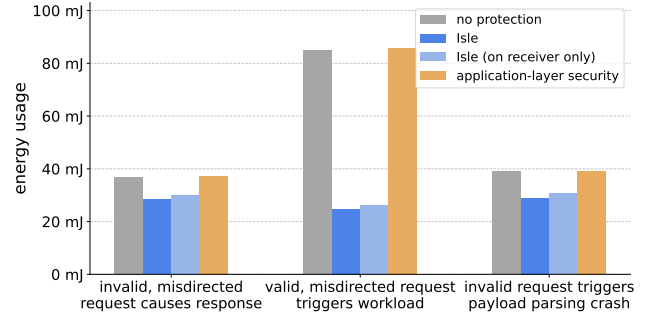


Figure 4: Comparison of energy usage of a device with and without Isle that receives a request in different scenarios in a one-minute span of time. Isle works to prevent an application from receiving traffic from outside of its realm, avoiding spending energy by not allowing the application to process the payload. By preventing even a properly-implemented application from sending a response, Isle reduces energy usage of the device by 23% compared to running the application un-isolated.

This experiment involves two applications running on two separate nRF52840 DK boards. The first is an audio classifier equipped with a CoAP server to allow on-demand classification of recently-captured audio. Upon receiving a request, the audio classifier exhibits one of three behaviors:

- properly responding to the invalid, misdirected request with an error message;
- mistakenly executing audio classification (due to misinterpretation);
- or encountering a fatal bug in handling the invalid request and crashing.

The second application, representing another application sharing the network, sends packets to the audio classifier, unintentionally interacting with it: such as a smart hub that auto-discovers the device and mistakes it for a kind of home automation device.

We observe each of the three aforementioned behaviors in four configurations: both applications with neither isolation nor security (“no protection”), both applications with application-layer security but no isolation (“application-layer security”), both applications running under Isle’s isolation in separate realms (“Isle”), and only the audio classifier application running under Isle’s isolation (“Isle (on receiver only)”). We measure the energy usage of the device running the audio classifier for one minute, during which time the second application sends a packet once. Figure 4 shows the total energy usage incurred in each scenario and configuration.

Even for an application that is functioning normally, the first scenario, there is significant benefit to energy-constrained devices. Being exposed to misdirected traffic causes the audio classifier application to expend a considerable 29.9% more energy when it is not isolated from the anomalous application. Receiving the packet, processing its contents, and transmitting a response consume energy, and Isle works to stop the processing and response from occurring unnecessarily.

In the second scenario, the misdirected request triggers the audio classification task on the receiving device. This could occur if the receiving application misinterprets the request as a valid instruction to perform some work. Applications can dynamically discover services offered by other applications on the network and interact with them, making this a plausible scenario as well. In this case, the interaction causes the device to consume more than $3\times$ the energy of the Isle-enabled configurations. Simply having a secure connection between the devices does not prevent the audio classifier from encountering this issue. This further demonstrates the utility of having the capability to isolate applications.

Finally, in the third scenario, the request causes the audio classifier application to crash. This could occur if the code parsing the request is not robust enough to handle unexpected requests: such as if a parser concludes the payload is larger than it actually is and performs an out-of-bounds access. In this case, the device must spend time and energy to restart the application and allow it to set itself up once again, consuming 34.6% more energy than the Isle-enabled configuration. Such fatal bugs not only incur energy usage to restart the application, but also affect the uptime of the application or can cause data loss (in this case, loss of the audio samples in memory).

Realm	Device	Network behavior
room	occupancy sensor	sends a multicast notification upon occupancy change
	lighting	listens for occupancy notifications from sensors
safety	smoke alarm	server exposes battery status
	smart lock	server exposes lock status
untrusted	smart speaker	periodically probes network for compatible devices every 5 minutes

Table 2: Networked device behavior setup. Shows the realm of each application and describes their behavior (Section 5.1.2).

5.1.2 Network of devices. To demonstrate the effectiveness of Isle’s isolation in a system that makes use of several devices and multiple realms, we deploy a collection of applications with varying network behaviors onto devices in the testbed network. This experiment measures the amount of time each device spends actively running the application over the course of three hours. The applications serve as those that one could reasonably find in a smart home environment, and they behave in different ways. An occupancy sensing application sends a multicast notification whenever the room occupancy status changes; we use three hours of ground truth data from an occupancy detection dataset [14] for realistic occupancy changes. A smart lighting application listens for occupancy notifications from sensors to determine when to turn lights on and off. A smoke alarm and a smart lock application expose their battery status and lock status, respectively, as a CoAP server. And finally, a smart

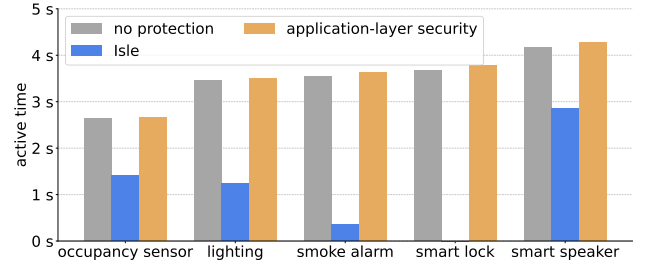


Figure 5: Total time applications spent running on devices in a Thread network over three hours with and without Isle. Without isolation, the occupancy sensor’s multicast notifications and the smart speaker’s probing requests increase the active time of applications, negatively affecting energy-constrained devices, like the door lock and smoke alarm. Isle isolates these applications into three groups and reduces their active time.

speaker sends requests to the .well-known/core resource [31] on devices in an attempt to auto-discover new, compatible devices.

We run the experiment with the applications having no protection, using software-based application-layer security, and using Isle for isolation and security. When running under Isle, we place the devices into three different realms: **room**, containing the occupancy sensor and lighting; **safety**, containing the smoke alarm and smart lock; and **untrusted**, containing only the smart speaker. Table 2 summarizes the application setup. We record the time that each device spent running their application and transmitting or receiving packets using each device’s timer peripheral and present the results in Figure 5.

In a network of unrestricted applications, the occupancy sensor and the probing smart speaker drive up the active time of all the devices in the network. The CoAP servers of the smart lock and smoke detector send responses to the probing smart speaker. The lighting application uses the notifications from the occupancy sensor but still receives unnecessary traffic from the smart speaker. And, again, there is little difference between having no protection and having application-layer security since it does not prevent the devices from interacting.

Enabling Isle on these devices and dividing them into domains (Table 2) restricts which applications can reach other applications, driving down the active time of the applications. All applications see a reduction in active time. The smart lock benefits the most, recording no time spent in its application code when shielded by Isle due to the occupancy sensor and smart speaker being in entirely separate realms.

5.2 Performance

Isle integrates with the OS and network stack to provide its isolation capability. We evaluate how it affects the performance of the network stack and as well as its impact on application network traffic.

5.2.1 Network stack. Because Isle is part of the network stack’s sequence of layers that processes packets, it increases the latency

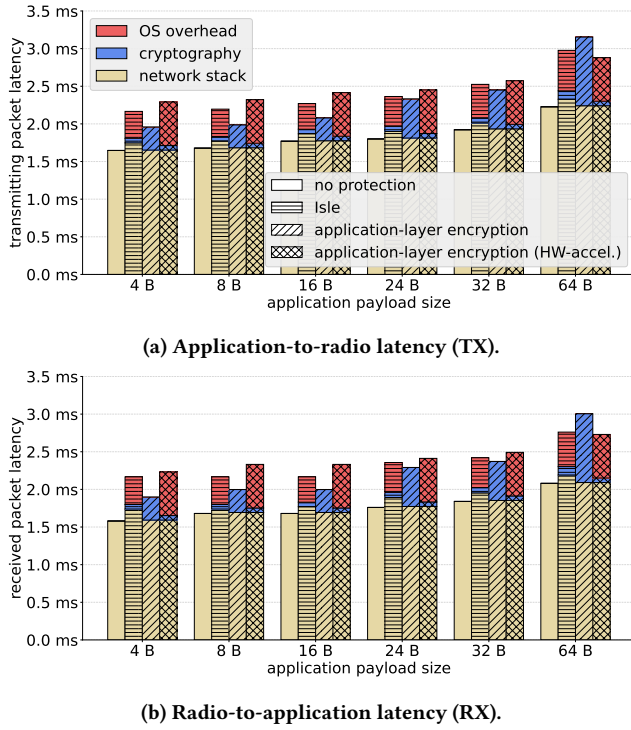


Figure 6: Latency of a single packet traversing the network stack. Securing communications adds processing time, but by re-using traditionally application-layer security to also provide isolation, Isle achieves latency similar to the hardware-accelerated application-layer security configuration. In some cases, Isle outperforms the hardware-accelerated application-layer alternative due to the OS overhead costs of using the encryption accelerator. The Isle kernel driver has lower-latency access to the accelerator, thus incurring lower overhead costs.

of every payload the application transmits and receives, which is a concern for latency-sensitive applications. We perform an evaluation to learn how much longer it takes payloads to traverse from the application to the radio and vice versa and also to understand where Isle spends that time compared to other approaches.

We deploy a test application on one nRF52840 DK which sends payloads ranging from 4 B to 64 B in size, to another nRF52840 DK. At points in the OS and network stack, we add GPIO pin-toggling instructions to record the duration spent performing different operations and break it down into three categories: **network stack**, the transformation of the payload to a packet; **cryptography**, the application of encryption, decryption, and MAC calculations; and **OS overhead**, the time spent interfacing with the OS (e.g., copying buffers, configuring the radio). Figure 6 provides a breakdown of the total time between the application and the radio. For transmissions (Figure 6a), this is the duration of time from the instant of the application’s send call to the instant the transmission completes. For receptions (Figure 6b), this is the duration of time from the instant the radio completes reception of the packet to the instant the application receives the contained payload.

In all cases, Isle adds less than one millisecond to the total network stack latency compared to fastest configuration, “no protection.” Overall, transmissions take slightly longer than receptions, and when transmitting a packet with an application-layer payload 64 B in length, Isle spends an additional 750 μ s to produce a packet. The additional processing that Isle performs to transform a CoAP message is visible as an increase in the network stack time. Isle’s total latency is more similar to the hardware-accelerated application-layer encryption configuration, and is slightly faster than it in all cases except with the 64 B packet case where it is slightly slower (+97 μ s). Interestingly, an application using its own software-based cryptography works faster than both Isle and the hardware-accelerated application-layer security for all cases except the 64-B case. The relatively small data size of the payload means the overhead of using the accelerator through the OS becomes a more significant expense on this evaluation platform.

Overall, this data can also be seen as a guide to what a developer can expect of switching from one system to another. Certainly, going from no protection to any of the other three approaches will add latency to processing a payload to or from a packet. Moving from software-based application-layer security to Isle will add a small amount of latency or have a similar latency cost. And moving from hardware-accelerated application-layer security to Isle will have little effect on the total latency.

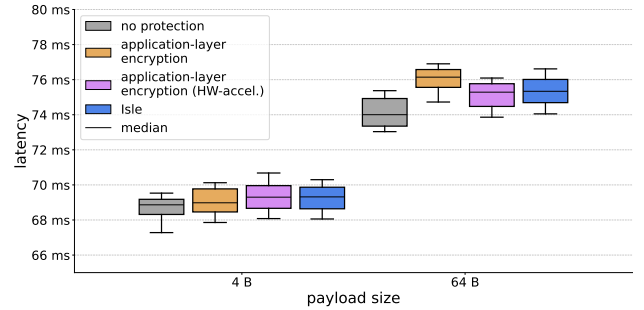


Figure 7: End-to-end latency of a device sending a single packet to another device. An application with Isle performs similarly to an application with application-layer security, showing that Isle maintains low overhead minimizes impact on application performance.

5.2.2 End-to-end single transmission latency. To understand the full latency impact that Isle has on communication between two networked applications, we study the end-to-end latency for transferring data over the network. This provides a more holistic view of the impact to the application. Not only do Isle and the other secure configurations have a processing latency in the network stack, but they also increase the number of bytes that the radio must transmit for cryptographic artifacts, such as MACs and IVs.

In this experiment, one nRF52840 DK acts as the sending device, and a second acts as the receiving device. The two devices are connected in the testbed by a third Thread router device. The application on the sending device transmits a single payload to the receiving device, and we record the duration of time between the

sending application making the call to send the payload and the receiving application receiving that same payload. We perform 30 trials of the single transmission and reception between the two devices in each of the security configurations. We collect data for 4-byte and 64-byte payloads, representing applications that occasionally send a small payload and applications that occasionally send a large payload. We record the end-to-end latency of these trials and present the results in Figure 7.

With a 4-byte payload, Isle performs similarly to the application-layer encryption configurations. Once again, this data shows that the software-based application-layer encryption slightly outperforms the secure alternatives, but the end-to-end latency difference compared to Isle is less than one millisecond. This shows that there is little difference in latency impact between using Isle and using application-layer security.

With a 64-byte payload, the differences between the configurations in end-to-end latency is more apparent. Isle still performs similarly to the hardware-accelerated application-layer configuration but the software-based application-layer security is now more often slower than Isle and the hardware-based alternative. Applications that use this large of a payload size would benefit from switching to using Isle.

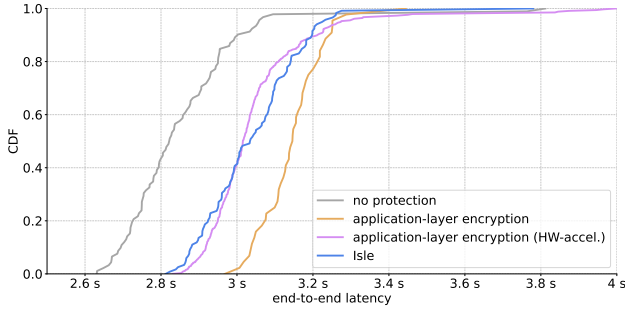


Figure 8: End-to-end latency of a device transferring 8 KB of audio samples to another device in a stream of packets under different configurations. Processing overhead and reduced packet space for application data decreases Isle’s performance relative to an application without protection, but performs similarly to an application with application-layer security without isolation.

5.2.3 End-to-end multi-packet latency. There are also many embedded applications that send a sequence of payloads to stream data from sensors or to send large data that exceeds the payload limit. We perform a similar end-to-end latency evaluation for this type of network communication. We again configure a sending application on an nRF52840 DK and another application on another nRF52840 DK to receive the payload. The sending application transmits 8 KB of audio data to the receiver as a sequence of payloads and the receiving application replies with acknowledgments, mirroring a real-world case requiring reliability. We use the GPIO of the sender and receiver to record the instant the sending application makes the call to send the first payload and the instant the receiving application successfully receives the last payload. We record the end-to-end latency of these trials and present a CDF in Figure 7.

The results are consistent with the single-payload evaluation (Section 5.2.2), but this evaluation more clearly shows the cost that the secure configurations add compared to each other and the unprotected alternative. Isle and the hardware-accelerated configuration respectively take, on average, 6.56% and 6.84% longer, to complete the transmission than the configuration with no protection in place. For this larger amount of data though, the software-based application-layer security configuration takes 10.20% longer. The additional cryptographic data that must accompany each payload extends the payload size, taking devices longer to transmit, receive, and process payloads and reducing the amount of space in each packet available for application data. Still, this evaluation shows that Isle performs similarly to hardware-accelerated application-layer security.

5.3 System overhead

Using Isle requires additional overheads that must be accounted for to understand its feasibility. We evaluate Isle with respect to its effect on application payload lengths, runtime memory usage, and code size.

5.3.1 Packet data overhead. Using cryptography to secure data in flight requires also sending cryptographic data (MAC, IV) to complete the decryption and verification of each packet. This unavoidable packet data overhead not only affects performance but also reduces the amount of data available in each packet for application data. Some cryptographic algorithms additionally impose padding constraints. Our evaluations in particular use AES-128-CCM, which the nRF52840 can accelerate with its on-chip peripheral, but it requires plaintext length to be a multiple of 8 bytes.

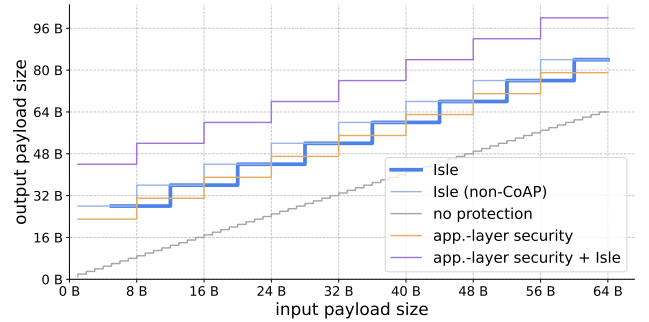


Figure 9: Payload size increase due to different security configurations. Isle (blue) imposes overhead similarly to application-layer security (orange) by reducing the plaintext that must be padded before encryption. By shifting responsibility for application-layer security out of the application and into the system, Isle avoids adding overhead equivalent to an entirely separate layer for isolation (purple).

We map these overheads in Figure 9, showing application payload input sizes and the corresponding output size after securing the payload. Whether hardware-accelerated or software-implemented, there is no difference in the output payload size for application-layer security, so a single line shows its overhead. The figure shows Isle’s packet data overhead both when securing a CoAP message

configuration	kernel + net. stack	app.	change
no protection	153.5 + 106.0 KB	62.1 KB	-
app.-layer sec.	153.5 + 106.0 KB	65.0 KB	+2.9 KB
Isle	157.0 + 107.0 KB	62.1 KB	+4.5 KB

Table 3: Code size overheads and net change in code size compared to the unmodified OS and no protection.

and when it is handling non-CoAP payloads. The smallest CoAP message size we consider is 5 bytes (4-byte header with a 1-byte token), so the plot for Isle starts at 5 B. Lastly, we provide the overhead for a payload protected by both application-layer security and Isle, representing the case of an entirely separate layer of security applied directly after the application-layer encryption.

When working with CoAP, Isle is sometimes able to provide isolation in addition to the protection of application-layer security with less packet data overhead than the application-layer security configuration. Since some data (header, options) from the CoAP message is not encrypted or only integrity-protected, it does not contribute to the plaintext length. This causes the plaintext to reach the padding threshold later, keeping the output payload size down. This shift in the overhead cost is due to Isle leveraging Group OSCORE, but still shows that by re-using application-layer security for isolation, Isle avoids extra packet data overhead. If the isolation layer were to be an additional layer on top of application-layer security, it would impose a higher packet data space cost, shown in purple. An isolation layer performing its own layer of encryption would have its own overhead for padding, a MAC, and an IV. This cost is even more than the packet data overhead that Isle imposes when it is not working with CoAP messages.

5.3.2 Code size overhead. The inclusion of Isle increases code size. However, due to its design, Isle shifts communication security from the application and into the underlying system, resulting in a decrease in application code size. The current implementation of OpenThread on Tock requires that the network stack run in userspace, typically as part of the application. To measure code size overhead, we build OpenThread as a separate application such that another application on the device wanting to access the network does so using interprocess communication. We build the application with a minimalistic configuration of the MbedTLS [37] library with only DTLS enabled and using software-based cryptography. This allows us to measure the code size changes for the kernel, network stack, and the application separately.

We present code size changes in Table 3. Using application-layer security increases the code size of the application binary (+2.9 KB). Shifting communication security responsibility into the system with Isle increases the code size of the kernel and network stack but allows the application to omit the application-layer security code for a net change +4.5 KB. Therefore, Isle has the larger code size overhead cost, but a multi-tenant device with a second application delegating application security to Isle would see a net savings (+4.5 KB for Isle < $+(2 \times 2.9)$ KB for two applications both implementing application-layer security). The evaluation devices have 1 MB of flash, meaning that they and other similar low-power microcontrollers can easily accommodate this change.

component	original	modified	change
kernel	39.2 KB	39.4 KB	+192 B
net. stack	48.9 KB	49.3 KB	+456 B
total	88.3 KB	88.6 KB	+648 B

Table 4: Memory overheads and net change in memory usage compared to the unmodified OS.

5.3.3 Memory overhead. The new kernel driver for Isle and the modified network stack require memory to function. This increases the memory requirements for running Isle. We provide the memory overhead totals and net increase values in Table 4. In addition to these fixed memory requirements, Isle requires 62 B of memory per realm an application is a member of to maintain the realm context. Compared to the 256 KB the nRF52840 has, this presents a small increase in memory usage which it can comfortably accommodate.

6 Discussion and Future Work

Cryptographic key management. With Isle, key management becomes an ability and responsibility of the network operator. Its design supports static or otherwise simple methods (e.g., manually distributing realm contexts) or more dynamic or automated approaches (e.g., controlling key distribution and rotation with a server). Key management and distribution itself is an active research area, even for wireless sensor networks [32], with different works providing trade-offs between security, resource cost, and scalability. A proposed standard for key management specifically for Group OSCORE [35] uses a central server.

Radio exhaustion DoS. Isle’s system-level design shields applications from aberrant packets, saving energy that would be consumed by further processing the packet. However, if the receiving device were able to avoid packet reception at the hardware layer, the device could spare even more resources. Without Isle present and working on both devices, or supposing the transmitting device is compromised, both devices expend unnecessary energy on the transmission and reception of the packet. In the worst case, the transmitting device can cause a denial-of-service condition due to the sheer volume of packets arriving. This is partially solvable with an extension to Isle’s approach. If routing nodes maintain information about which realms nodes are members of, they can drop packets not bound for a valid destination. This “distributed filtering” requires realm membership data be distributed among routers in some manner but does not resolve cases in which the nodes are in the same realm or are neighbors. Resolving these cases would provide a valuable research contribution.

7 Conclusion

Convergence on a unified mesh networking protocol presents both great opportunity and challenges for embedded devices. While it has the power to streamline the growth of smart ecosystems and the development of powerful, new applications, it also erodes the natural boundaries that once protected constrained devices from the interactions of unrelated applications, threatening the sustainable, stable operation of these devices. Effective, network-level isolation is a

requirement for enabling scalable, multi-vendor networks of constrained devices. Without it, noisy, fragile IoT infrastructure is an inevitability. Isle provides network-level isolation for applications to re-enforce those boundaries. By subsuming application-layer security into the underlying operating system, Isle is able to shield constrained devices from the processing costs and stability risks of unwanted network traffic, supporting multi-tenant operation while also maintaining sensitivity to the compute and networking constraints of the low-power embedded system.

Acknowledgments

We thank the anonymous reviewers; their feedback improved this paper. This work is supported by the National Science Foundation under Grant Nos. 2144940 and 2534083, as well as the Semiconductor Research Corporation (SRC) under project HWS-3127.001.

References

- [1] [n. d.]. *OpenThread*. Retrieved January 31, 2026 from <https://openthread.io>
- [2] [n. d.]. *What Is Constrained Application Protocol?* Retrieved January 28, 2026 from <https://www.akamai.com/glossary/what-is-constrained-application-protocol-coap>
- [3] 2020. *Apple introduces HomePod mini: A powerful smart speaker with amazing sound*. Retrieved January 13, 2026 from <https://www.apple.com/ca/newsroom/2020/10/apple-introduces-homepod-mini-a-powerful-smart-speaker-with-amazing-sound/>
- [4] 2020. *An Introduction to Thread*. Retrieved January 27, 2026 from <https://blog.nordicsemi.com/getconnected/an-introduction-to-thread>
- [5] 2022. *KNX IOT on Thread: New Open Source Stack and Hardware Devkit Boost Product Development*. Retrieved January 13, 2026 from <https://threadgroup.org/Newsroom/Blog/KNX-IOT-ON-THREAD-NEW-OPEN-SOURCE-STACK-AND-HARDWARE-DEVKIT-BOOST-PRODUCT-DEVELOPMENT>
- [6] 2025. *How Matter and Thread Make Your Home More Helpful*. Retrieved January 13, 2026 from <https://store.google.com/intl/en/ideas/articles/matter-thread-for-your-smart-home/>
- [7] 2025. *KNX IoT for Smart Buildings*. Retrieved January 13, 2026 from <https://www.nxp.com/company/about-nxp/smarter-world-blog/BL-KNX-IOT-SMART-BUILDINGS>
- [8] 2025. *nRF52840 DK*. Retrieved April 29, 2025 from <https://www.nordicsemi.com/Products/Development-hardware/nRF52840-DK>
- [9] Iqbal Alam, Kashif Sharif, Fan Li, Zohaib Latif, M. M. Karim, Sujit Biswas, Boubakr Nour, and Yu Wang. 2020. A Survey of Network Virtualization Techniques for Internet of Things Using SDN and NFV. *ACM Comput. Surv.* 53, 2, Article 35 (April 2020), 40 pages. doi:10.1145/3379444
- [10] Amazon Web Services. 2026. *Implementing LPWAN IoT Solutions with CoAP*. Technical Report. Amazon Web Services, Inc. <https://docs.aws.amazon.com/whitepapers/latest/implementing-lpwan-solutions-with-aws/implementing-lpwan-iot-solutions-with-coap.html> Part of the "Implementing Low-Power Wide-Area Network (LPWAN) Solutions with AWS IoT" whitepaper series.
- [11] Bluetooth SIG. 2023. *Mesh Profile Specification v1.1*. Bluetooth Special Interest Group. <https://www.bluetooth.com/Adopted> on September 19, 2023.
- [12] Conner Bradley and David Barrera. 2023. Escaping Vendor Mortality: A New Paradigm for Extending IoT Device Longevity. In *Proceedings of the 2023 New Security Paradigms Workshop* (Segovia, Spain) (NSPW '23). Association for Computing Machinery, New York, NY, USA, 1–16. doi:10.1145/3633500.3633501
- [13] Peter Bull, Ron Austin, Evgenii Popov, Mak Sharma, and Richard Watson. 2016. Flow Based Security for IoT Devices Using an SDN Gateway. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*. 157–163. doi:10.1109/FiCloud.2016.30
- [14] Luis Candanedo. 2016. Occupancy Detection. UCI Machine Learning Repository. DOI: <https://doi.org/10.24432/C5X01N>.
- [15] Center for Internet Security. 2024. *CIS Critical Security Controls Version 8.1*. Center for Internet Security, East Greenbush, NY. <https://www.cisecurity.org/controls/v8-1>
- [16] Connectivity Standards Alliance. 2024. *Matter Core Specification v1.3*. Connectivity Standards Alliance. <https://csa-iot.org/developer-resource/specifications-download-request/> Document 23-27349.
- [17] Carmen Delgado, María Canales, Jorge Ortín, José Ramón Gállego, Alessandro Redondi, Sonda Bousnina, and Matteo Cesana. 2018. Joint Application Admission Control and Network Slicing in Virtual Sensor Networks. *IEEE Internet of Things Journal* 5, 1 (2018), 28–43. doi:10.1109/JIOT.2017.2769446
- [18] Sinh Do, Luong-Vy Le, Bao-Shuh Paul Lin, and Li-Ping Tung. 2019. SDN/NFV-Based Network Infrastructure for Enhancing IoT Gateways. In *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. 1135–1142. doi:10.1109/iThings/GreenCom/CPSCom/SmartData.2019.00192
- [19] Christos Efstratiou, Ilias Leontiadis, Cecilia Mascolo, and Jon Crowcroft. 2010. A shared sensor network infrastructure. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*. 367–368.
- [20] Mo Haghighi and Dave Cliff. 2013. Sensomax: An agent-based middleware for decentralized dynamic data-gathering in wireless sensor networks. In *2013 International Conference on Collaboration Technologies and Systems (CTS)*. 107–114. doi:10.1109/CTS.2013.6567214
- [21] International Organization for Standardization. 2022. *Information security, cybersecurity and privacy protection – Information security management systems – Requirements*. Technical Report ISO/IEC 27001:2022. International Organization for Standardization, Geneva, Switzerland. <https://www.iso.org/standard/27001>
- [22] Isam Ishaq, Jeroen Hoebeke, Ingrid Moerman, and Piet Demeester. 2012. Internet of Things Virtual Networks: Bringing Network Virtualization to Resource-Constrained Devices. In *2012 IEEE International Conference on Green Computing and Communications*. IEEE Computer Society, Los Alamitos, CA, USA, 293–300. doi:10.1109/GreenCom.2012.152
- [23] Anura P Jayasumana, Qi Han, and Tissa H Illangasekare. 2007. Virtual sensor networks-a resource efficient approach for concurrent applications. In *Fourth International Conference on Information Technology (ITNG'07)*. IEEE, 111–115.
- [24] Imran Khan, Fatna Belqasmi, Roch Glitho, Noel Crespi, Monique Morrow, and Paul Polakos. 2016. Wireless sensor network virtualization: A survey. *IEEE Communications Surveys & Tutorials* 18, 1 (2016), 553–576. doi:10.1109/COMST.2015.2412971
- [25] Ilias Leontiadis, Christos Efstratiou, Cecilia Mascolo, and Jon Crowcroft. 2012. SenShare: transforming sensor networks into multi-application sensing infrastructures. In *European Conference on Wireless Sensor Networks*. Springer, 65–81.
- [26] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) (SOSP'17). ACM, New York, NY, USA, 234–251. doi:10.1145/3132747.3132786
- [27] Mike Ojo, Davide Adami, and Stefano Giordano. 2016. A SDN-IoT Architecture with NFV Implementation. In *2016 IEEE Globecom Workshops (GC Wkshps)*. 1–6. doi:10.1109/GLOCOMW.2016.7848825
- [28] OMA SpecWorks. 2024. *Lightweight Machine to Machine Technical Specification: Core*. Technical Specification OMA-TS-LightweightM2M_Core-V1_2_2-20240613-A. Open Mobile Alliance. <https://www.openmobilealliance.org>
- [29] Raspberry Pi Ltd. 2015. Raspberry Pi 2 Model B. <https://www.raspberrypi.com/Computer-Hardware>.
- [30] Göran Selander, John Preuß Mattsson, Francesca Palombini, and Ludwig Seitz. 2019. Object Security for Constrained RESTful Environments (OSCORE). RFC 8613. doi:10.17487/RFC8613
- [31] Zach Shelby, Klaus Hartke, and Carsten Bormann. 2014. The Constrained Application Protocol (CoAP). RFC 7252. doi:10.17487/RFC7252
- [32] Sabina Szymboniak. 2024. Key Distribution and Authentication Protocols in Wireless Sensor Networks: A Survey. *ACM Comput. Surv.* 56, 6, Article 144 (Jan. 2024), 31 pages. doi:10.1145/3638043
- [33] Texas Instruments. 2015. *INA219 Zero-Drift, Bidirectional Current/Power Monitor With I2C Interface*. Texas Instruments, Dallas, TX. <https://www.ti.com/lit/ds/symlink/ina219.pdf> SBOS448G.
- [34] Thread Group, Inc. 2024. *Thread 1.4.0 Specification*. Thread Group, Inc. <https://threadgroup.org/ThreadSpec> Available upon request from the Thread Group.
- [35] Marco Tiloca and Francesca Palombini. 2026. *Key Management for Group Object Security for Constrained RESTful Environments (Group OSCORE) Using Authentication and Authorization for Constrained Environments (ACE)*. Internet-Draft draft-ietf-ace-key-groupcomm-oscore-21. Internet Engineering Task Force. <https://datatracker.ietf.org/doc/draft-ietf-ace-key-groupcomm-oscore-21/> Work in Progress.
- [36] Marco Tiloca, Göran Selander, Francesca Palombini, Rikard Höglund, and Jiye Park. 2025. *Group Object Security for Constrained RESTful Environments (Group OSCORE)*. Internet-Draft draft-ietf-core-oscore-groupcomm-28. Internet Engineering Task Force. <https://datatracker.ietf.org> Work in Progress.
- [37] Trusted Firmware. 2024. Mbed TLS. <https://github.com/Mbed-TLS/mbedtls> Version 3.6.0.
- [38] Elena Uchiteleva, Abdallah Shami, and Ahmed Refaey. 2017. Virtualization of Wireless Sensor Networks Through MAC Layer Resource Scheduling. *IEEE Sensors Journal* 17, 5 (2017), 1562–1576. doi:10.1109/JSEN.2016.2643670
- [39] Yang Yu, Loren J Rittle, Vartika Bhandari, and Jason B LeBrun. 2006. Supporting concurrent applications in wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*. 139–152.