# A DTLS Abstraction Layer for the Recursive Networking Architecture in RIOT

M. Aiman Ismail, Thomas C. Schmidt
Internet Technologies Group, Dept. Informatik, HAW Hamburg, Germany
{muhammadaimanbin.ismail, t.schmidt}@haw-hamburg.de

***Abstract*—On the Internet of Things (IoT), devices continuously communicate with each other, with a gateway, or other Internet nodes. Often devices are constrained and use insecure channels for their communication, which exposes them to a selection of attacks that may extract sensitive pieces of information or manipulate dialogues for the purpose of sabotaging.**

**This paper presents a new layer in the RIOT networking architecture to seamlessly integrate secure communication between applications using DTLS. The layer acts as a modular abstraction layer of the different DTLS implementations, enabling swapping of the underlying implementation with just a few lines of code. This paper also introduces credman, a new module to manage credentials used for (D)TLS connections.**

## I. Introduction

Security is an important part when communicating through the Internet. Despite the fact that without proper security practices, bad actors could break into our network infrastructures and cause severe damage to parties involved, there are still numerous devices, IoT appliances in particular, that expose themselves on the Internet without having any proper security measures in place.

Datagram Transport Layer Security (DTLS) [1] is a protocol for traffic encryption on top of UDP [2]. It is based on the concepts of TLS [3] and provides equivalent security guarantees. DTLS guarantees reliable transport during the handshake process but maintains UDP transport properties during application data transfer. The protocol is deliberately designed to be as similar to TLS as possible, both to minimize new security inventions and to maximize the amount of code and infrastructure reuse.

RIOT [4] is an open source real-time OS, based on a modular architecture built around a lightweight micro-kernel, and developed by a worldwide community of developers. The modular approach enables easy prototyping and development to test new ideas and deploy applications. Its default network stack GNRC follows a cleanly layered, recursive design that easily allows for stacking and exchanging protocol layers or implementations.

In this paper, we describe how we built the DTLS abstraction layer on top of existing components in the RIOT networking architecture. This layer provides an API that can be implemented using third-party DTLS libraries. It is designed to be independent of the underlying DTLS implementation, therefore allows the DTLS stack to be exchanged without altering the applications that uses it. We also introduce a new RIOT module *credman* to manage the credentials used for the handshake.

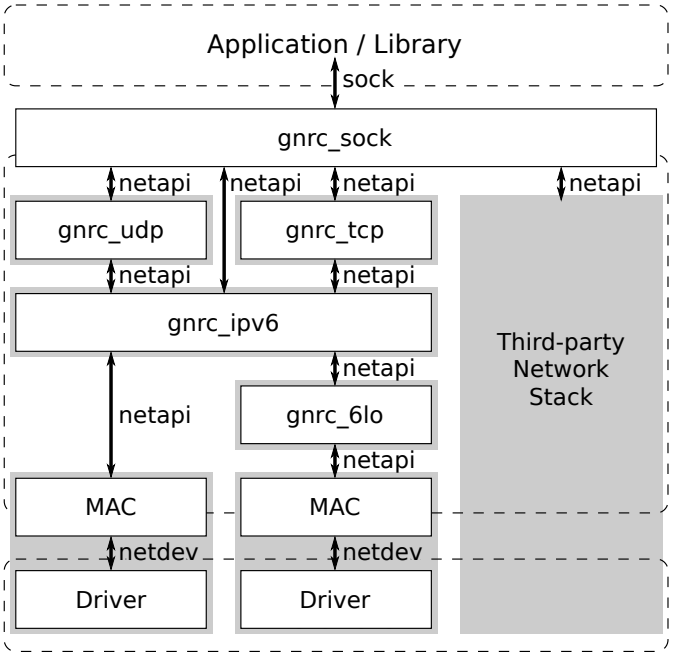


Fig. 1: RIOT networking stack

The remainder of this paper is structured as follows. In Section II, we introduce the existing networking stack of RIOT. In Sections III, we describe the new secure network stack, and Section IV presents experiments that assess its performance. In Section V, we draw conclusions with an outlook on future work.

## II. RIOT networking subsystem

The RIOT networking subsystem is designed to follow a modular architecture with clean interfaces for abstracting all layers [5]. This facilitates the creation and integration of new protocols, different implementations, or additional layers such as a new encryption layer to the existing stack. It consists of the two external APIs *netdev, sock*, and a single internal API for communication between layers, *netapi*. It is noteworthy that the RIOT networking subsystem simultaneously supports multiple interfaces with different protocol stacks, which makes it capable of running gateway services. An architectural overview is visualized in Figure 1.

*The Device Driver API: netdev*. Individual network devices in RIOT are abstracted via *netdev*, which allows networking stacks access to the devices via a common, portable interface. *netdev* remains neutral in that it does not enforce implementation details regarding memory allocation, data flattening, and threading. These decisions are delegated to the users of the interface.

*The Internal Protocol Interface: netapi*. Internal protocol layers in the RIOT networking subsystem can be recursively composed via the *netapi*. The interface is kept simple so that even an exotic networking protocol could be implemented against it. Message passed between layers are typed as following: two asynchronous message types (`MSG_TYPE_SND`, `MSG_TYPE_RCV`) and two synchronous message types (`MSG_TYPE_GET`, `MSG_TYPE_SET`) that expects a reply in form of `MSG_TYPE_ACK` typed message. No further semantic are built into the messages of *netapi*, but certain preconditions on packets or option values handed to *netapi* can be set as requirements to implement more complex behavior that goes beyond these plain specification.

*The User Programming API: sock*. This module provides a network API for applications and libraries in RIOT. It provides a set of functions to establish connections or send and receive datagrams using different types of protocols. In comparison to POSIX sockets, *sock* does not require complex and memory expensive implementation and therefore more suited for use in constrained hardware. Only common type and definitions from either *libc* or POSIX. This ensures that *sock* is easy to port to other target OS.

*GNRC* is the native IPv6 networking stack for RIOT. It takes full advantage of the multi-threading model supported by RIOT to foster a clean protocol separation via well-defined interfaces and IPC. Each network protocol is encapsulated in its own thread and uses RIOT thread-targeted IPC with a message queue in each thread to communicate between layers. Other stacks that introduce different networking protocols such as ICN also integrate via the same interfaces. Various experimental evaluations and benchmarks [5], [6] have proven the feasibility and efficiency of this flexible approach to networking in RIOT.

## III. Introducing the Secure Network Stack

The modular nature of the existing GNRC stack allows for an easy extension by adding DTLS at the top while maintaining its modularity. We introduced two new modules *credman* and *sock_dtls* (see Figure 2).

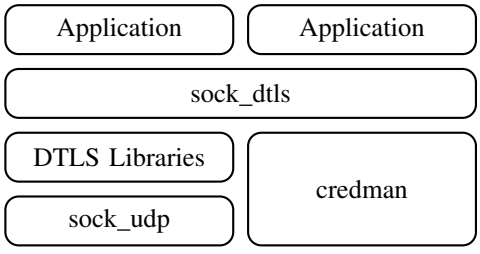


Fig. 2: Architecture of sock_dtls

*credman* is a module to manage credentials used in (D)TLS encryption protocols. Credentials registered with the system are identified by using the tuple of int-based tag *credman_type_t* and the credential type *credman_tag_t*. This in combination with the *sock_dtls* API allows users to register multiple credentials of the same type, which can be the case if the nodes are communicating with multiple other nodes simultaneously and each node uses different credentials for authentication.

*credman* does not copy the credentials into the system memory. It only has information about the credentials and points to the location of the credential itself, which can be stored in protected regions of the memory. Users will have to ensure that a credential is available at the location given to credman during the lifetime of their application.

We defined a new *sock* type — *sock_dtls*. It is designed to mimic the behavior of *sock_udp* as closely as possible so that integrating it into existing applications and libraries can be done without introducing too many new changes. By adding a line in the Makefile, users can choose which underlying DTLS implementation to use. Swapping to a new DTLS implementation is simply done by specifying the corresponding implementation in the Makefile. Through this mechanism, testing and evaluation of DTLS implementations can be performed without altering the application.

Figure 2 summarizes the integration of the DTLS abstraction layer with existing network stack in RIOT. Currently, RIOT only has support for tinyDTLS[1] but there is ongoing work[2] to add support for wolfSSL[3].

The use cases of *sock_dtls* are twofold, the DTLS server and the DTLS client. The server is created by `sock_dtls_create()`. Then it needs to tell the credentials to sock by `sock_dtls_register_credential_tags()`. After the call `sock_dtls_init_server()` the server is ready to receive new DTLS session establishment requests from clients.

DTLS clients also need to create the sock using `sock_dtls_create()` and then register the credentials with `sock_dtls_register_credential_tags()`. After that, a session to a DTLS server can be established using `sock_dtls_establish_session()`. If successful, the session can be used to send and receive datagram packet like in a normal UDP channel.

For the DTLS server, the operations can be summarised as follows:

1) Create *sock_dtls*
2) Register credentials available for use
3) Initialize the server
4) Start listening for incoming datagram packets

[1]https://projects.eclipse.org/projects/iot.tinydtls
[2]https://github.com/RIOT-OS/RIOT/pull/10308
[3]https://www.wolfssl.com

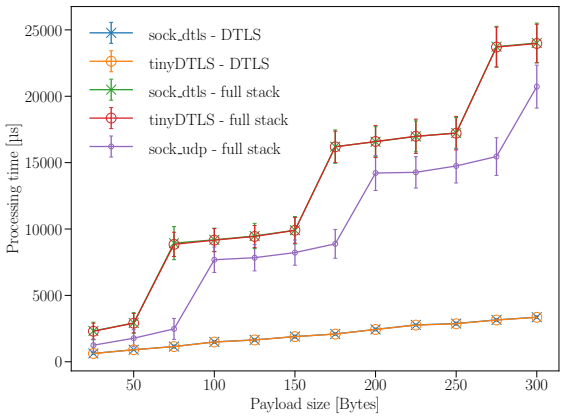


Fig. 3: CPU Overhead

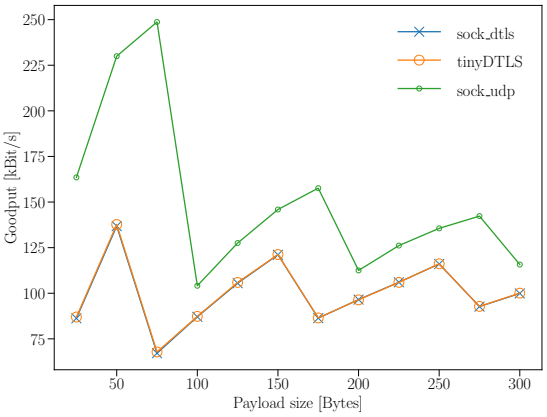


Fig. 4: Average Goodput

As for the DTLS client, the first two steps are the same as for the server followed by

3) Establish session with a DTLS server
4) Start sending and receiving datagram packets

## IV. Experiments and Evaluation

We are now ready to validate our concept and assess the performance of our implementation. We compared *sock_dtls* with tinyDTLS and *sock_udp* and examine the metrics CPU overhead and goodput during the transmission of payloads as well as memory consumption. All measurements were performed on *samr21-xpro* boards positioned side-by-side over the 802.15.4 wireless radio network [7] with 6LoWPAN encapsulation and header compression [8].

We wrote three versions of a client and a server program that send packets of increasing payload sizes from the client to the server while recording the time taken to transmit the packets. The first version uses tinyDTLS directly while the second version uses our new DTLS abstraction layer *sock_dtls* with tinyDTLS. The third version employs no encryption layer but plainly uses RIOT UDP sock API *sock_udp* to transmit the packets and acts as a controlling baseline.

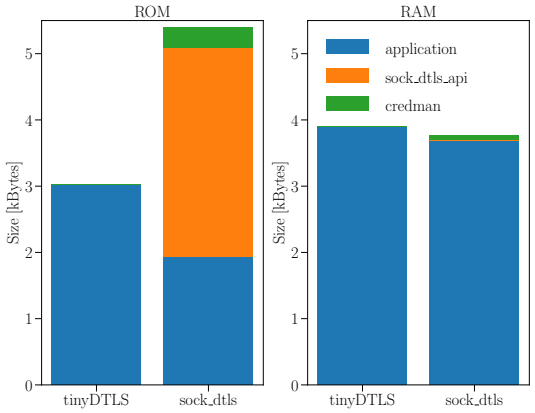


Fig. 5: Memory Usage

The test was setup as follows unless stated otherwise. The server is instantiated to listen for new connections and receives packet from clients. For each received packet, the payload size is logged into a file. On the client side, two metrics are measured: the time taken to process a packet for the full network stack, that is (1) DTLS, UDP, IP, 6LoWPAN, MAC, and auxiliary components, and (2) the time taken to process only the DTLS part of the transmission, which starts from accepting the packet from user and encrypting it using specified keying materials to just before passing it to UDP layer for further processing. The test is run with payload size ranging between 25 Bytes and 300 Bytes in 25 Bytes intervals. Each configuration is repeated 5000 times with averages and standard deviations recorded in the following diagrams.

**CPU Overhead**. Figure 3 depicts the CPU overhead during packet transmission. The test program using *sock_dtls* and tinyDTLS need approximately the same average processing time per packet with *sock_dtls* being slightly higher. The extra overhead when adding an abstraction layer is expected as a tradeoff for faster prototyping time and ease of use, which in this case is virtually negligible. The step-like line shaped for the full stack processing can be attributed to the fragmentation of packets by the underlying 6LoWPAN layer when certain size limits are reached. Comparison of the times taken to process only the DTLS layer shows an almosts linear line of the processing time with increasing payload size and again, there is only little difference between the values.

This clearly indicates that our *sock_dtls* abstraction layer comes at negligible processing overhead.

**Goodput**. The average goodput is shown in in Figure 4. It follows the same trend with the *sock_dtls* version admitting approximately the same performance values as the tinyDTLS version. These results not only indicate a picture consistent with processing, but also confirm the robustness of our interface layer.

**Memory**. Figure 5 compares the memory consumptions of the different DTLS code versions. Here we measured the memory usage of a simple echo client and server application implemented using *sock_dtls* and tinyDTLS instead of our test application to mirror a more complete DTLS application compared to the test application. The hardware setup is the same.

The RAM usage of both programs is similar with *sock_dtls* saving around 120 Bytes compared to tinyDTLS. This saving is mainly contributed by the compiler, which can optimize away some of the variables used for the sending and receiving functions in user application but not in tinyDTLS. As a result, even though we actually need about 80 Bytes more in *sock_dtls* for *credman* and the API, we still end up using less RAM. Nevertheless, because the saving is only around 100 Bytes and is mainly caused by compiler optimization, we could actually say that the RAM usage is approximately the same in both versions and the exact value is determined by the quality of implementation in user application.

In contrast, the ROM usage in *sock_dtls* is about two kilobytes larger than in tinyDTLS. The larger ROM size is due to the code size of *sock_dtls*. This value is implementation specific as each implementation needs to be implemented against the DTLS sock interface first before used as the underlying implementation of *sock_dtls*. When using tinyDTLS specifically, we could actually delegate the bulk of credential management to *credman*. For tinyDTLS this must be implemented as callbacks by the users. This simplifies the user application and actually achieves about the same performance using less code.

## V. Conclusion and Future Works

In this paper, we introduced and analyzed the new DTLS abstraction layer designed to be modular and easy for integrating into existing applications. We demonstrated that the tradeoff between performance and ease of use is well acceptable for normal use cases. Leveraging a clean and implementation-independent interface, we increased the portability of applications and also the maintainability of upper layer protocol implementations such as CoAP [9] over time.

In the future, we will work on implementing a DTLS profile for authentication and authorization for the constrained environment such as [10] to provide a framework for a secure network infrastructure. The integration of *sock_dtls* in upper layer protocols such as the RIOT *gcoap*[4]is also on our schedule.

## References


[1] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," IETF, RFC 6347, January 2012.

[2] J. Postel, "User Datagram Protocol," IETF, RFC 768, August 1980.

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

[4] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, December 2018. [Online]. Available: http://dx.doi.org/10.1109/JIOT.2018.2815038

[5] M. Lenders, P. Kietzmann, O. Hahm, H. Petersen, C. Gündogan, E. Baccelli, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Connecting the World of Embedded Mobiles: The RIOT Approach to Ubiquitous Networking for the Internet of Things," Open Archive: arXiv.org, Technical Report arXiv:1801.02833, January 2018. [Online]. Available: https://arxiv.org/abs/1801.02833

[6] C. Gündogan, P. Kietzmann, M. Lenders, H. Petersen, T. C. Schmidt, and M. Wählisch, "NDN, CoAP, and MQTT: A Comparative Measurement Study in the IoT," in *Proc. of 5th ACM Conference on Information-Centric Networking (ICN)*. New York, NY, USA: ACM, September 2018, pp. 159–171. [Online]. Available: https://conferences.sigcomm.org/acm-icn/2018/proceedings/icn18-final46.pdf

[7] IEEE 802.15 Working Group, "IEEE Standard for Local and metropolitan area networks—Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)," IEEE, New York, NY, USA, Tech. Rep. IEEE Std 802.15.4™–2011, Sep 2011.

[8] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks," IETF, RFC 4944, September 2007.

[9] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," IETF, RFC 7252, June 2014.

[10] S. Gerdes, O. Bergmann, C. Bormann, G. Selander, and L. Seitz, "Datagram Transport Layer Security (DTLS) Profile for Authentication and Authorization for Constrained Environments (ACE)," IETF, Internet-Draft – work in progress 01, March 2017.


[4]https://riot-os.org/api/group__net__gcoap.html