Evaluieren von Embedded-Systemen

WireGuard oder DTLS – ein Vergleich

(Bild: tong2530/stock.adobe.com)

Um einen Code auf Mikrocontrollern zu evaluieren, gibt es verschiedene Möglichkeiten. Besonders bekannt ist hierbei die DTLS-Verbindung, für die es bereits zahlreiche Implementierungen gibt. Jedoch lohnt sich ein Blick auf WireGuard – kann dieses Protokoll DTLS ausstechen?

Von Ruien Karimi

Industrie 4.0 verändert die Produktion in Unternehmen aufgrund der zunehmenden Vernetzung hochverfügbarer Systeme. Dadurch ähneln Industriegeräte herkömmlichen Computern und erfordern vergleichbare Sicherheitsmethoden. Allerdings eignen sich die gängigen Verfahren eines Rechenzentrums nicht für vernetzte IoT-Geräte: IoT-Geräte weisen vor allem in eingebetteten Systemen (ES) begrenzte Ressourcen wie Rechenleistung und Speicher auf. So sind herkömmliche Security-Konzepte mit hohem Leistungsbedarf nicht anwendbar. Nötig sind Sicherheitsapplikationen mit niedrigem Ressourcenbedarf [1].

»WireGuard« ist ein hochperformantes VPN-Protokoll mit schlankem Quell-

code [2]. Aus diesem Grund sollte WireGuard auf einem tief eingebetteten System (DES) wenig Ressourcen in Anspruch nehmen und sich für sichere Vernetzung eignen. Im Folgenden wird die Eignung von WireGuard für DES unter Verwendung einer Datagram-Transport-Layer-Security(DTLS)-Verbindung als Benchmark untersucht, da DTLS mit der leichtgewichtigen kryptografischen Bibliothek »mbedTLS« ebenfalls geringen Ressourcenbedarf aufweist [3], [4].

Als DES wurde der Mikrocontroller »STM32f767zi« mit einer Taktfrequenz von bis zu 216 MHz und einem Flash-Speicher von 2 MB und 512 KB SRAM verwendet. Als Entwicklungsumgebung wurde »CubeIDE« eingesetzt mit mbedTLS als Middleware, was die Implementierung der DTLS-Verbindung vereinfachte. Wie WireGuard eignet sich DTLS für Echtzeit-Applikationen mit geringer Latenz. Es setzt auf dem User Datagram Protocol (UDP) auf und hat einen geringeren Overhead als eine TLS-Verbindung [5].

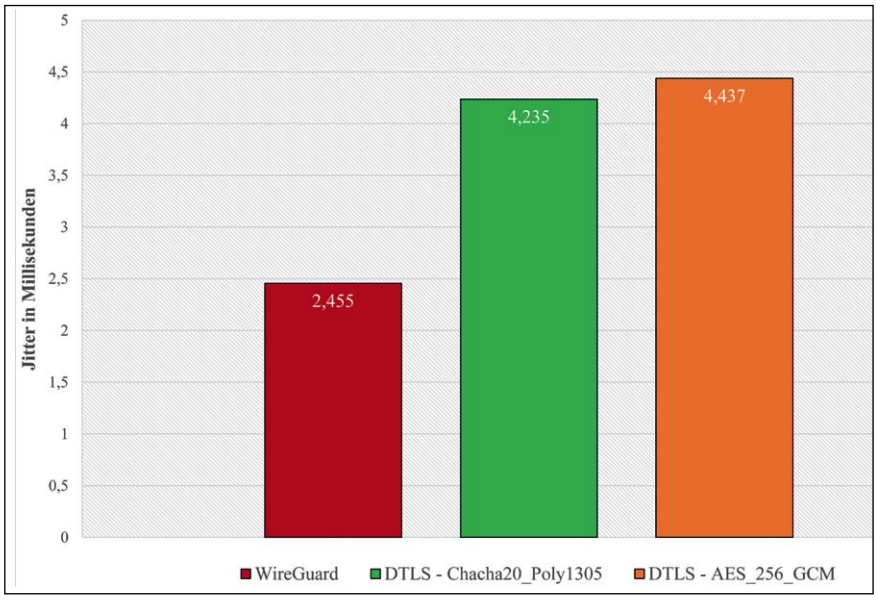


Bild 1. Messergebnisse der Jitter-Werte für die WireGuard- und DTLS-Verbindung. (Bild: Ingenics digital)

Methodik: Implementieren von WireGuard

Die verwendete WireGuard-Implementierung stellt eine C-Implementierung des WireGuard-Protokolls für die Verwendung mit dem »LwIP«-Stack zur Verfügung. Sie ist urheberrechtlich geschützt und unter »BSD 3-Clause« lizenziert (Urheberrecht Daniel Hope [6]). Sie eignet sich für ES wegen der überschaubaren Größe des Quellcodes, der geringen Speicheranforderungen in Bezug auf Stack-Größe, Flash und RAM-Speicher sowie der Kompatibilität mit dem LwIP-Stack, der sich über die CubeIDE gut einbinden lässt [6].

Die Implementierung von Daniel Hope verwendet die »RAW«-Programmierschnittstelle (API) und wurde ausschließlich für die Verwendung des LwIP-Stacks ohne Betriebssystem entwickelt. Für die vorliegende Untersuchung musste die WireGuard-Implementierung in die Umgebung des STM32-Mikrocontrollers mit »FreeRTOS« und LwIP-Stack integriert werden. Weil die RAW-API als nicht »threadsafe« gilt, wurden die Funktionen, welche die RAW-API verwenden, meist durch die »Netconn«-API ersetzt. Für Funktionen, die sich nicht für die Netconn-API umschreiben lassen, musste man Kern und API vor gleichzeitigen Zugriffen durch Semaphoren und Mutexe schützen.

Hierzu wurde die Start-Handshake-Funktion im WireGuard-Code durch die Netconn-API mit »Netbufs« und Freigabe des globalen TCPIP-Mutex angepasst: Die Funktion *UNLOCK_TCPIP_CORE* gibt den globalen Mutex *lock_tcpip_core* frei. Dieser binäre Semaphor kann vom Client-Code gesperrt werden, um LwIP-Operationen durchzuführen, ohne in den TCPIP-Thread mit Callbacks wechseln zu müssen. Hierbei kann der Client-Code den Mutex sperren, um den Zugriff auf den LwIP-Protokollstack zu schützen und Race Conditions zu verhindern. Sobald die Operationen abgeschlossen sind, kann der Client-Code den Mutex wieder freigeben und dem TCPIP-Thread die Kontrolle zurückgeben – der Client-Code ist in dem Fall der WireGuard-Code [7].

Die Initialisierungsfunktion von WireGuard startet einen LwIP-Timer, der nach einer bestimmten Zeit immer wieder vom TCPIP-Thread aufgerufen wird, der den LwIP-Prozess steuert und regelmäßig die Funktion *sys_check_timeouts* aufruft, die ausstehende Timer-Ereignisse im TCPIP-Stack verarbeitet. Zuvor ruft der Thread *LOCK_TCPIP_CORE* auf und sperrt *lock_tcpip_core* während des gesamten LwIP-Prozesses, zum Beispiel beim Start des Handshake von WireGuard. Beim Senden der ersten Handshake-Nachricht wird später im Sendevorgang ebenfalls versucht, denselben Mutex *lock_tcpip_core* zu sperren. Weil diese binäre Semaphore bereits gesperrt ist, tritt ein Deadlock auf, womit das Senden fehlschlägt. Um das Problem zu lösen, gibt man vor jeder Sendefunktion, die von WireGuard-Code aufgerufen wird, den Mutex *lock_tcpip_core* für das Senden der Nachricht frei. Außerdem ist eine kleine Anpassung am LwIP-Stack nötig, um unendliche Rekursionen beim Senden von Paketen durch die WireGuard-Sende-Funktionen zu vermeiden.

Da die WireGuard-Schnittstelle eine virtuelle Netzwerkschnittstelle ist, muss man die Checksummen von STM32 per Software berechnen – standardmäßig berechnet sie der STM32 per Hardware.

Bei der Funktion *wireguard_tai64n_now* in »wireguard-platform.c« von Daniel Hope tritt ein Problem auf: WireGuard fügt einen »TAI64N«-Zeitstempel zum Vermeiden von Replay-Angriffen in der ersten Nachricht ein. Der WireGuard-Server überwacht den größten empfangenen Zeitstempel pro Client und verwirft Pakete, deren Zeitstempel kleiner oder gleich des letzten

empfangenen Zeitstempels des WireGuard-Clients sind. Sobald der Client sich nach einem Neustart wieder mit dem Server verbinden möchte, muss der Client einen größeren Zeitstempel verwenden als vor dem Neustart [2]. Der STM32, der als WireGuard-Client dient, erfüllt die Bedingung mit der bereitgestellten Funktion *wireguard_tai64n_now* nicht.
Der STM32 könnte alle 5 Minuten seinen Zeitstempel in den Flash schreiben. Nach einem Neustart des STM32 würde er den gespeicherten Zeitstempel plus 5 Minuten verwenden. Somit gewährleistet der STM32, dass der Zeitstempel immer größer ist als der zuvor gespeicherte Wert und den Anforderungen der WireGuard-Implementierung entspricht. Allerdings wäre der damit verbundene Aufwand unverhältnismäßig hoch. Stattdessen wurde als Abhilfe auf dem Raspberry Pi ein Shell-Skript geschrieben, das die WireGuard-Verbindung zwischen dem STM32 und dem Rasperry Pi überwacht. Weil alle 25 s mindestens ein »Keepalive«-Paket zu senden ist, startet der WireGuard-Server nach 30 s neu, falls es in dem Zeitraum zu keinem Datenaustausch zwischen den Kommunikationspartnern gekommen ist. Hierbei setzt der WireGuard-Server den größten empfangenen Zeitstempel pro Client zurück. Das Zeitintervall von 25 s kann je nach Konfiguration des WireGuard-Clients variieren.

WireGuard-Messung

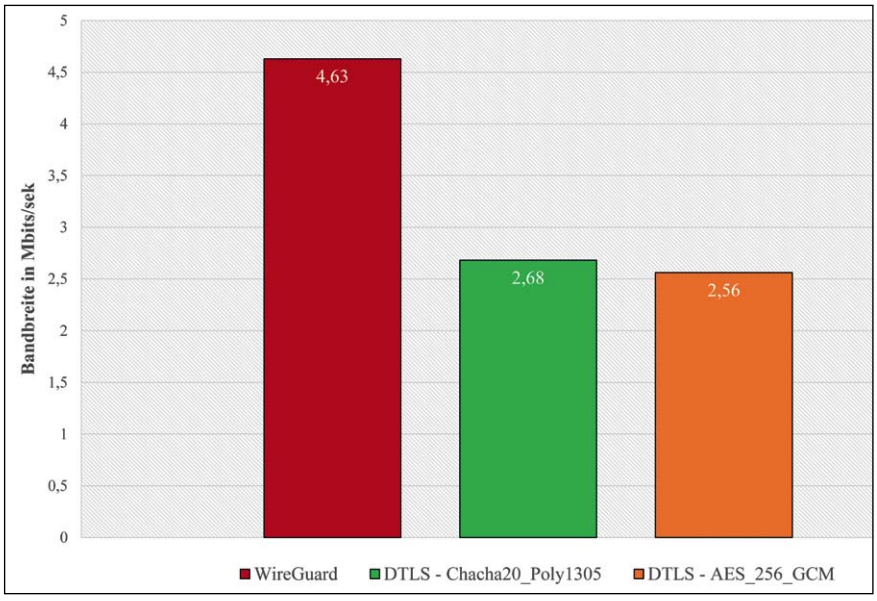
Die Bestimmung des Netzwerkdatendurchsatzes zwischen WireGuard-Client und Server erfolgt mit dem OpenSource Tool »iperf«. Es basiert auf dem Client-Server-Modell und misst die maximale Netzwerkbandbreite zwischen Client und Server von TCP und UDP. Beim Messen des Datendurchsatzes sendet der iperf-Client so schnell wie möglich UDP-Pakete an den iperf-Server. Der iperf-Server führt die Messung anhand der Anzahl der empfangenen Pakete pro Zeiteinheit durch – das Messen des Netzwerkdatendurchsatzes mit iperf liefert Werte für die Netzwerkbandbreite, Jitter und den Datentransfer über einen bestimmten Zeitraum. Hierbei erfolgt die Messung alle 5 s über einen Zeitraum von 60 s.
Beim Messen des Ressourcenbedarfs auf dem STM32-Mikrocontroller bestimmt man die CPU-Auslastung und den maximalen Stack-Speicherverbrauch der FreeRTOS-Tasks mit der FreeRTOS-Funktion *vTaskGetRunTimeStats* [8]. Der Build-Analyzer von CubeIDE ermittelt den verwendeten Flash- und RAM-Speicher des STM32, den FreeRTOS, LwIP und WireGuard benötigen. Um die Daten besser vergleichen zu können, werden bei der WireGuard- und DTLS-Implementierung FreeRTOS und LwIP identisch konfiguriert.

Implementieren von DTLS

Der DTLS-Client wird mit »mbedTLS« (Middleware aus CubeIDE) auf dem STM32 implementiert. Mit CubeIDE wird der TLS-Server-Code deaktiviert, da er hier nicht erforderlich ist, was Flash-Speicher spart.
Die bereitgestellten Schlüssel und Zertifikate sind öffentlich zugänglich, daher ist deren Sicherheit für kommerzielle Zwecke nicht ausreichend, für Testzwecke aber durchaus – deshalb wurden sie für diese Untersuchung eingesetzt. Die vorhandenen Entropiequellen reichten aber nicht aus, da die Standardeinstellung von mbedTLS eine softwarebasierte Random-Number-Generator(RNG)-Variante ohne eigene Entropiequelle vorsieht. Zur Gewährleistung ausreichender Entropie muss RNG mit der Hardware-Abstraction-Layer(HAL)-Bibliothek und dem zugehörigen Interrupt in CubeIDE aktiviert werden. Weiterhin sind Anpassungen für eine alternative HW-basierte Entropiequelle mit starken Zufallsdatenquellen nötig:

- Deaktivieren von *MBEDTLS_NO_DEFAULT_ENTROPY_SOURCE*
- Aktivieren von *MBEDTLS_ENTROPY_FORCE_SHA256*
- Aktivieren der Option *MBEDTLS_ENTROPY_HARDWARE_ALT*

Hinzufügen von *mbedtls_hardware_poll* in »entropy.c« stellt die Generierung zufälliger Zahlen mit HAL-RNG sicher. Alternativ kann die Überprüfung der Entropie deaktiviert werden, auf Kosten der Sicherheit.
Das DTLS-Protokoll wurde angepasst, um die Sendung beschädigter DTLS-Pakete zu unterbinden, die vom DTLS-Server nicht verarbeitet werden: Die Option *MBEDTLS_MEMORY_BUFFER_ALLOC_C* wurde aktiviert und konfiguriert, um die benutzerdefinierte Speicherverwaltung in einem statisch angelegten Buffer (65 KB) zur Speicherallokation für DTLS-Operationen [9] zu ermöglichen. Die ausgewählte DTLS-Implementierung wurde auf dem STM32 integriert und ist unter der Apache License Version 2.0 verfügbar [10].



(Bild: Ingenics digital)

Bild 2. Messergebnisse der Netzwerkbandbreite für die WireGuard- und DTLS-Verbindung.

DTLS-Messung

Zum Vergleich der WireGuard-Implementierung mit der DTLS-Verbindung wurden vergleichbare Messverfahren angewendet: Netzwerkdatendurchsatz und der Ressourcenverbrauch auf dem Mikrocontroller.

Zur Bestimmung des Netzwerkdatendurchsatzes wurde ebenfalls iperf verwendet. Dazu lauscht der iperf-Server auf einen bestimmten Port. Da das DTLS-Protokoll im Gegensatz zu WireGuard eine Punkt-zu-Punkt-Verbindung ist, muss das entschlüsselte Paket vom Zielport der DTLS-Verbindung zum iperf-Port weitergeleitet werden. Dafür wurde »Goldy« auf dem Raspberry Pi installiert, der als Netzwerk-Server arbeitet. Goldy ist ein leichtgewichtiger DTLS-Proxy, der ebenfalls die mbedTLS-Bibliothek verwendet und als Vermittler zwischen dem Client und dem eigentlichen Server fungiert. Goldy ist wie ein DTLS-Tunnel und übernimmt die Funktion eines DTLS-Servers [11].

Als Cipher Suite wurde *TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256* für die Vergleichsmessung zur WireGuard-Verbindung festgelegt. Zum einen basieren *ECDHE_RSA* (bei DTLS) und das DH-Verfahren mit *Curve25519* (bei WireGuard) auf dem Diffie-Hellman-Verfahren mit elliptischen Kurven zur Aushandlung eines gemeinsamen geheimen Schlüssels. Zum anderen verwenden beide Verbindungen *ChaCha20Poly1305* für die symmetrische Datenverschlüsselung. So lassen sich DTLS und WireGuard besser vergleichen. Als zusätzliche Referenz für die Netzwerkbandbreite wurde eine zweite Messung mit der Cipher Suite *TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384* durchgeführt.

Analog zur WireGuard-Messung wurde die CPU-Auslastung und der maximale Stack-Speicherverbrauch der einzelnen FreeRTOS-Tasks durch die FreeRTOS-Funktion *vTaskGetRunTimeStats* bestimmt. Außerdem ermittelte der Build-Analyzer von CubeIDE den vom STM32 verwendeten Flash- und RAM-Speicher, den FreeRTOS, LwIP und DTLS mit der Middleware mbedTLS benötigen. Die Speicherkonfiguration von FreeRTOS und LwIP sind identisch zum WireGuard-Projekt. Der Unterschied besteht durch die zusätzliche Middleware mbedTLS und den statischen Buffer der Größe 65.000 Byte.

Tabelle 1: Ressourcenverbrauch der WireGuard- und DTLS-Verbindungen auf dem STM32

Ressourcenverbrauch	WireGuard	DTLS
RAM-Speicher	110,43 KByte	180,52 KByte
Flash-Speicher	170,49 KByte	466,43 KByte
maximaler Stackspeicherverbrauch[1]	1840 Byte	3824 Byte
CPU-Auslastung - Initialisierung[2]	12,41 %	69,02 %
CPU-Auslastung - laufender Betrieb[2, 3]	2,27 %	2,29 %

[1] Der WireGuard-Thread alleine hat einen maximalen Stackspeicherverbrauch von 860 Byte. Bei WireGuard wird ein weiteres Netzwerkinterface (wg0) hinzugefügt. Dies führt beim TCPIP-Thread zu einer Erhöhung des Stackspeicherverbrauchs um 980 Byte. Aus diesem Grund wurde dieser Wert in der Angabe der Tabelle mitberücksichtigt.
[2] Durchschnittswert aus 3 Messperioden, welche die CPU-Auslastung der letzten 5 Sekunden erfassen.
[3] In jeder Messperiode von 5 Sekunden wurde ein Paket gesendet.

Evaluierung der Messergebnisse: Netzwerkdatendurchsatz

Die Bandbreite misst die maximale Datenmenge, die über eine Verbindung pro Zeiteinheit übertragen werden kann. Der Datendurchsatz beschreibt die Datenmenge, die in einem bestimmten Zeitraum übertragen wurde. Jitter gibt die Varianz der Laufzeit von Datenpaketen in ms an. Je größer die Jitter-Werte sind, desto größer ist die Latenz der übertragenen Datenpakete.

Bild 1 stellt die Jitter-Werte der WireGuard-Verbindung und der DTLS-Verbindung gegenüber. Für die Messung der DTLS-Verbindung sind die symmetrischen Verschlüsselungsverfahren angegeben, damit erkennbar ist, welche Cipher Suite verwendet wurde. WireGuard weist mit 2,455 ms einen nahezu doppelt so guten Wert wie die DTLS-Verbindung auf, wobei die Jitter-Werte der DTLS-Verbindung mit 4,337 ms (orange) und 4,235 ms (grün) nahe beieinander sind.

Bild 2 zeigt die Netzwerkbandbreite in Mbit/s zwischen WireGuard und der DTLS-Verbindung. Mit 4,63 Mbit/s zeigt WireGuard eine signifikant bessere Netzwerkbandbreite als die DTLS-Verbindung mit 2,68 Mbit/s (grün) und 2,56 Mbit/s (orange).

Bild 3 zeigt den Datendurchsatz in MB pro Minute zwischen der WireGuard-Verbindung und der DTLS-Verbindung. Er beträgt bei WireGuard 33,1 MB pro Minute, also ein nahezu doppelt so großer Wert wie die DTLS-Verbindung mit 19,2 MB pro Minute (grün) und 18,3 MB (orange).

Evaluierung der Messergebnisse: Ressourcenbedarf

Tabelle 1 zeigt den RAM- und Flash-Speicher auf dem STM32 für das WireGuard- und DTLS-Projekt. WireGuard weist dabei im Vergleich zu DTLS sowohl im RAM- als auch im Flash-Speicher geringere Werte auf. Die Werte liegen im erwarteten Bereich, da DTLS die zusätzliche Middleware mbedTLS erfordert, die sich deutlich auf den Flash-Speicher auswirkt. Der statisch angelegte Buffer von 65 kB, der für DTLS-Operationen verwendet wird, wird im RAM alloziert und erhöht den RAM-Speicher. Zudem benötigt der WireGuard-Thread einen niedrigeren Stack-Speicher als der DTLS-Thread.

Die Messergebnisse bestätigen, dass die ausgewählte WireGuard-Implementierung vergleichsweise geringe Speicheranforderungen in Bezug auf Stack-, Flash- und RAM-Speicher hat. Die Aufzeichnung der CPU-Auslastung

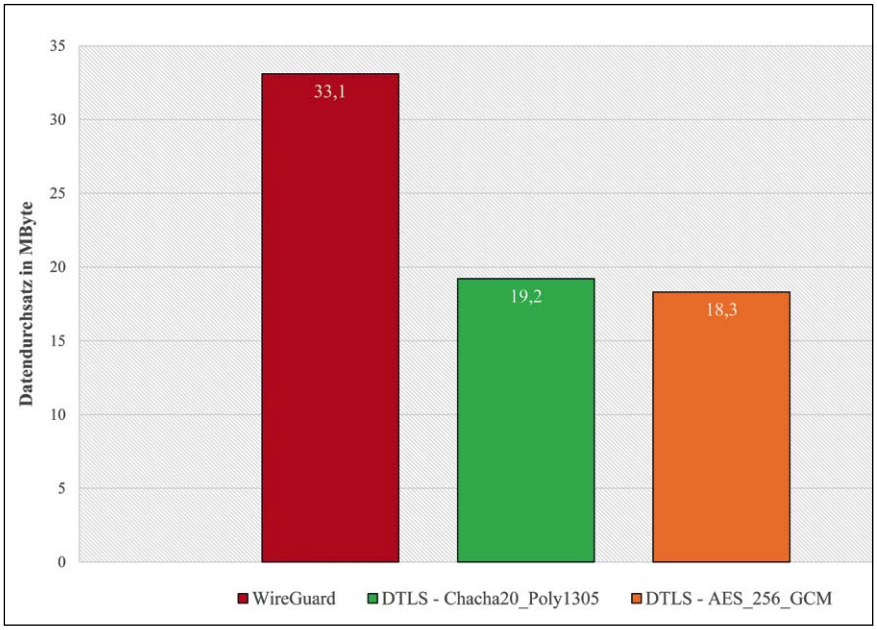


Bild 3. Messergebnisse des Datendurchsatzes über einen Zeitraum von 60 s der WireGuard- und DTLS-Verbindung. (Bild: Ingenics digital)

wird bei der Initialisierung zeitgleich mit dem Start des WireGuard/DTLS-Threads gestartet. Während die CPU-Auslastung im laufenden Betrieb bei WireGuard und DTLS nahezu identisch ist, geht aus Tabelle 1 hervor, dass das DTLS-Projekt eine deutlich höhere CPU-Auslastung bei der Initialisierung hat als das WireGuard-Projekt. Das ist darauf zurückzuführen, dass DTLS im Vergleich zu WireGuard einen umfangreichen Initialisierungsprozess erfordert, was ebenfalls den höheren Stack-Speicherbedarfs des DTLS-Threads erklärt.

WireGuard ist für tief eingebettete Systeme geeignet

Die Messergebnisse bestätigen die Eignung von WireGuard für DES. Im Vergleich zum DTLS-Protokoll, das für Echtzeit-Applikationen mit geringer Latenz und datenintensiven Anforderungen bekannt ist, zeigt Wire-Guard eine bessere Leistung: Es hat im Vergleich zur DTLS-Verbindung niedrigere Jitter-Werte und eine höhere Bandbreite. So ermöglicht WireGuard einen effizienteren Datendurchsatz bei geringerem Ressourcenbedarf auf dem STM32. Somit eignet sich WireGuard für die Integration in DES. Die Kombination aus geringem Ressourcenbedarf und besserer Performance macht WireGuard zu einem geeigneten und für viele Zwecke sogar besseren Tool als DTLS für DES.

Es ist wichtig anzumerken, dass die vorliegenden Messungen WireGuard lediglich mit dem DTLS-Protokoll vergleichen. Weitere Vergleiche mit anderen VPN-Applikationen wie OpenVPN oder IPsec können zusätzliche Einblicke liefern und eine umfassendere Evaluierung der Eignung von WireGuard für DES ermöglichen. Laut WireGuard-Whitepaper gibt es bereits Messungen, die eine bessere Performance von WireGuard im Vergleich zu OpenVPN und IPsec [7] zeigen. Die Vergleiche wurden jedoch nicht speziell auf DES bezogen und sind möglicherweise nicht direkt auf deren Kontext übertragbar. Zudem ist zu klären, wie sich das WireGuard-Protokoll auf verschiedenen Mikrocontrollern mit unterschiedlichen Ausstattungen verhält. Eine Untersuchung der Optimierungsmöglichkeiten der gemessenen Parameter wie Speicherbedarf und Datendurchsatz auf anderen Mikrocontrollern wäre von Interesse. Hierdurch könnte man ein vertieftes Verständnis dafür gewinnen, wie WireGuard in Bezug auf Leistung und Ressourcenbedarf auf verschiedenen Mikrocontrollern funktioniert. ts

Literatur

[1] Embedded-Systeme gegen Hacker absichern. R. Witzgall. 2020. https://www.security-insider.de/embedded-systeme-gegen-hacker-absichern-a-86c783397f2e1976e6b0a37cdd570ee3/. Aufgerufen am 10.08.2023 um 15:15.

[2] WireGuard-Protokoll. J.A. Donenfeld. 2022. https://www.wireguard.com/protocol/. Aufgerufen am 10.08.2023 um 15:17.

[3] mbed TLS. DeWiki. 2023. https://dewiki.de/Lexikon/Mbed_TLS. Aufgerufen am 10.08.2023 um 15:21.

[4] Mbed TLS. D. Rodgman. 2023. https://developer.trustedfirmware.org/w/mbed-tls/. Aufgerufen am 10.08.2023 um 15:22.

[5] Was ist UDP. D.-I. F. S. Luber. https://web.archive.org/web/20221228200620/. Aufgerufen am 10.08.2023 um 15:26.

[6] WireGuard Implementation for IwIP, D. Hope. 2023. https://github.com/smartalock/wireguard-lwip. Aufgerufen am 10.08.2023 um 15:30.

[7] Core locking and MPU. L.W. Adam Dunkels. 2023. https://www.nongnu.org/lwip/2_0_x/group__lwip__opts__lock.html. Aufgerufen am 10.08.2023 um 15:33.

[8] Run Time Statistics. FreeRTOS. https://www.freertos.org/rtos-run-time-stats.html. Aufgerufen am 10.08.2023 um 15:39.

[9] Alternative ways of allocating memory in Mbed TLS. T.M.T. Contributors. 2023. https://mbed-tls.readthedocs.io/en/latest/kb/how-to/using-static-memory-instead-of-the-heap/. Aufgerufen am 10.08.2023 um 15:44.

[10] Mbed-TLS. M. TLS-Projekt. 2023. https://github.com/Mbed-TLS/mbedtls/blob/development/programs/ssl/dtls_client.c. aufgerufen am 10.08.2023 um 15:46.

[11] Goldy. I. S. Innovation. 2023. https://github.com/ibm-security-innovation/goldy/. Aufgerufen am 10.08.2023 um 15:48.

Ruien Karimi

ist Werkstudent bei Ingenics digital. Er hat Elektro- und Informationstechnik mit dem Schwerpunkt technischer Informatik studiert und verfügt über gute Kenntnisse in den Bereichen Embedded Systems sowie sicherer Netzwerkkommunikation zwischen IoT-Geräten.
E-Mail: sales@ingenics-digital.com