On TLS 1.3
Early Performance Analysis in the IoT Field

Simone Bossi[1], Tea Anselmo[2] and Guido Bertoni[2]
[1]*Department of Computer Science, Università degli studi di Milano, Milan, Italy*
[2]*STMicroelectronics, Via Olivetti 2, Agrate B.za, Italy*

Keywords: TLS 1.3, Authenticated Encryption, IoT, Performance Analysis, Data Security.

Abstract: The TLS 1.3 specifications are subject to change before the final release, and there are still details to be clarified, but yet some directions have been stated. In the IoT scenario, where devices are constrained, it is important and critical that the added security benefits of the new TLS 1.3 does not increase complexity and power consumption significantly compared to TLS 1.2. This paper provides an overview of the novelties introduced in TLS 1.3 draft finalized to improve security and latency of the protocol: the reworked handshake flows and the newly adopted cryptographic algorithms are analyzed and compared in terms of security and latency to the current TLS in use. In particular, the analysis is focused on performance and memory requirements overhead introduced by the TLS 1.3 current specifications, and the final section reports simulation results of a commercial cryptographic library running on a low end device with an STM32 microcontroller.

1 INTRODUCTION

The emerging technology of the Internet of Things (IoT) is having a huge impact on the generated data volume, on network traffic and the way of handling them. We are experiencing an extraordinary diffusion of smartphones and tablets, of new devices such as wearables, environmental sensors, smart-home systems. They are all part of a vast set of devices requiring a connection - possibly even to the Internet - for purposes of data communication, remote monitoring, management, and accounting. In this complex and intricate context of technologies and protocols, it is unanimously recognized that security is a key enabler for success of IoT and for end user acceptance. In particular, securing the communications that involve constrained low-end devices is an hard task since the enhancements of security network protocols always introduce additional workload and memory requirements.

The paper focuses on the TLS (*Transport Layer Security*) protocol because it is probably the most popular security network protocol in the plethora of existing protocols, and because a new version, TLS 1.3 (Rescorla, 2015), is under development (still a draft) to respond to security issues and needs. TLS is implemented on top of the transport layer and used to secure a wide range of TCP/IP protocols as for example HTTP (Rescorla, 2000), FTP (Ford-Hutchinson, 2005), SMTP (Hoffman, 2002), and XMPP (Saint-Andre, 2011).

The main purpose of the TLS protocol is to provide communication security in terms of privacy and authenticity between hosts communicating over insecure channels. In order to achieve this goal TLS provides three sub-protocols: handshake, record and alert protocols. We will focus our analysis on handshake and record layers which make use of cryptographic primitives.

This paper will treat the main novelties introduced in the TLS 1.3 draft and discussed by the TLS working group in the mailing list, focusing on performance and memory requirements of constrained devices, and supporting the analysis with test results.

The reminder of this paper is organized as follows. Section 2 compares the message flows of TLS 1.2 and 1.3 handshake protocols explaining the main differences proposed in the draft. Section 3 gives an overview on the record layer focusing on Authenticated Encryption problematic, current solutions, and its adoption in TLS. Section 4 summarizes the state of the art security requirements for algorithms employed in the protocol in terms of key sizes. In Section 5 and 6 we present and comment our work of analysis and benchmark respectively on TLS 1.2 handshake and TLS 1.3 additional overhead. Finally Section 7 de-

Bossi, S., Anselmo, T. and Bertoni, G.
On TLS 1.3 - Early Performance Analysis in the IoT Field.
DOI: 10.5220/0005688901170125
In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP 2016)*, pages 117-125
ISBN: 978-989-758-167-0

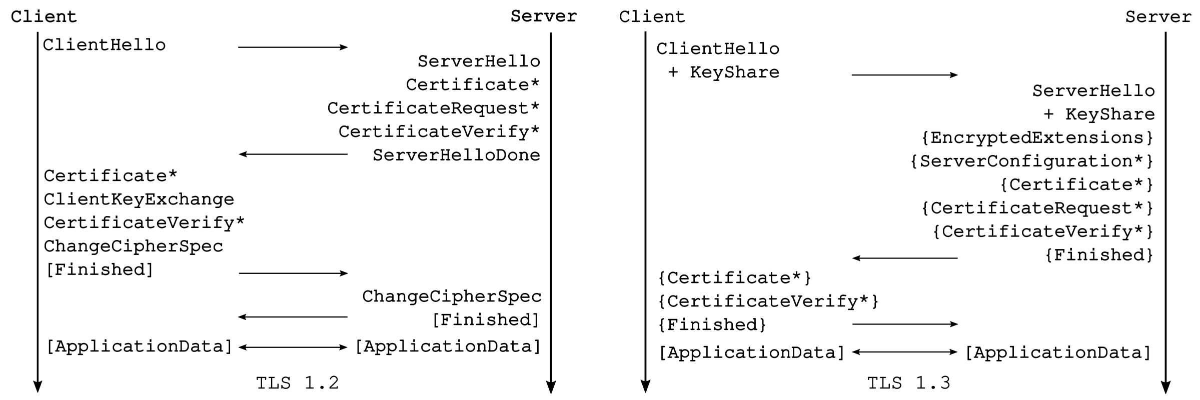

Figure 1: Message flow comparison between TLS 1.2 and 1.3 for a full handshake. $+$ indicates extensions sent in the previously noted message, $*$ indicates optional or situation-dependent messages, $\{\}$ indicates messages protected using keys derived from the ephemeral secret, and $[]$ indicates messages protected using keys derived from the master secret.

scribes use cases for TLS on IoT devices motivating the choice of the discussed algorithms in different situations and gives some concluding remarks.

2 TLS HANDSHAKE PROTOCOL

The handshake protocol determines the agreement of cryptographic algorithms, the mutual setting of common cryptographic parameters, including the keys that will be used to encrypt the communication and to authenticate the involved parties. Beside traditional purposes of the TLS protocol (Dierks and Rescorla, 2008), which are cryptographic security, interoperability and extensibility, the main goals defined for the new TLS version 1.3 concern the reduction of the n-RTT (number of Round Trip Time) required for the handshake, and the encryption of the sensitive information exchanged in this phase. Figure 1 provides a comparative view of the TLS 1.2 handshake and the novel TLS 1.3 scheme as is in current protocol draft. The new proposed handshake flow provides a 1-RTT full handshake solution against the 2-RTT of the previous version. The rearranged flow includes the transmission of the client key parameters during its first flight within the new extension KeyShare, thus anticipating the shared key computation on the server side just after its KeyShare packet. At this point, the server can already compute the shared secret (or *premaster secret*), derive the Ephemeral Secret (*ES*) and start to encrypt all remaining handshake traffic. Furthermore the *ServerConfiguration* packet will be introduced for the purpose of enabling 0-RTT exchange upon subsequent connections. However our performance analysis is focused on the full handshake which represents the worst case.

Among the new security features proposed in the draft (Rescorla, 2015), it's also worth noting that all non-ephemeral key exchange algorithms (including RSA) have been removed from the cipher suites. The aim here is to provide *Perfect Forward Secrecy* (PFS), that guarantees the agreed key will not be compromised even when agreed keys derived from the same long-term keying material in a subsequent run are compromised.

All these security enhancements necessarily comes with an increased workload for both the client and the server. In Section 5 we present our analysis methodology and results for the handshake protocol.

3 TLS RECORD PROTOCOL

The record protocol takes as input uninterpreted data from higher layers, fragments it into blocks of maximum 2^{14} bytes, and applies the symmetric encryption algorithm as established in the initial handshake. In TLS 1.3, as opposed to previous versions, all symmetric ciphers are modeled as *"Authenticated Encryption with Additional Data"* (McGrew, 2008). In the rest of this section we summarize the properties of authenticated encryption and the solutions developed over the time.

3.1 Authenticated Encryption

In the context of communication over unsecure channels, the need for privacy and authenticity led to the rise of scrappy combination of separate encryption and message authentication techniques. As a result the confusion in the user community brought to a number of security break, as for example the break of WEP protocol in 802.11 (Borisov et al., 2001).

To overcome these flaws there have been various attempts to give a standardized way to combine *Message authentication codes* (MAC) and ciphers, in order to provide *Authenticated Encryption* (AE) through a "generic composition" design. The main approaches are *MAC-and-Encrypt* (M&E), *MAC-then-Encrypt* (MtE), and *Encrypt-then-MAC* (EtM). In M&E, adopted by the SSH protocol (Ylonen and Lonvick, 2006), the original message is encrypted and authenticated through the MAC function, and the concatenation of ciphertext and authentication tag are given as output. MtE instead appends authentication tag to the plaintext and then encrypt the whole. This is the approach used by TLS 1.2 and prior (Dierks and Rescorla, 2006), (Dierks and Rescorla, 2008). Finally EtM differs from the previous since it does not authenticate the plaintext, but, as the name suggests, it first encrypts the message, then calculates the authentication tag over the ciphertext and outputs the concatenation of ciphertext and tag. Old versions of the IPSec protocol implement EtM ((Kent and Atkinson, 1998)). A detailed analysis on the security of authenticated encryption schemes designed by generic composition is presented in (Bellare and Namprempre, 2000). The paper acknowledges the EtM as the only secure strategy against both privacy and integrity attacks providing the proof for this case, and counter-examples for M&E and MtE. This analysis has been also confirmed by (Krawczyk, 2001), and a series of attacks has exploited the weaknesses of MtE such as (Al Fardan and Paterson, 2013),(Möller et al., 2014), (Degabriele and Paterson, 2010), and M&E (Bellare et al., 2004).

Alongside with these threats that afflicts AE ciphers built on the generic composition paradigm, the need to handle associated-data when using AE modes became clear. This is particularly useful in network communications where there are non-sensitive data (e.g. IP addresses, headers, etc.) that does not require privacy protection, but rather authentication. In (Rogaway, 2002) this problematic is formalized and named *authenticated-encryption with associated-data* (AEAD). The paper investigates the importance and the goals of AEAD ciphers, providing a provable-security treatment. Furthermore solutions to the problem of translating an AE-scheme to an AEAD-scheme are presented.

3.2 Authenticated Encryption in TLS

Even if AEAD ciphers are allowed (Salowey et al., 2008), as previously mentioned the TLS protocol version 1.2 mandates an MtE construction based on AES128-CBC cipher and HMAC authentication. Since this strategy has been proved not to be secure over the time ((Bellare and Namprempre, 2000), (Krawczyk, 2001), (Al Fardan and Paterson, 2013),(Möller et al., 2014), (Degabriele and Paterson, 2010)) the TLS working group has provided a way to negotiate the use of EtM mechanism via specification in *HelloExtensions* (Gutmann, 2014).

In the new TLS version 1.3 the intent is to provide authenticated encryption only via AEAD ciphers. At the time of writing, the main symmetric ciphers included are AES operated in the *Galois Counter Mode* (GCM) (McGrew and Viega, 2004)(Dworkin, 2007), and Chacha20-Poly1305 (Nir and Langley, 2015) which is a construction of Chacha20 cipher (Bernstein, 2008) and Poly1305 message authentication code (Bernstein, 2005). Chacha20-Poly1035 AEAD cipher has been employed since 2014 by some famous and widely used web services like Google, CloudFlare and openSSH claiming its high throughput and security. In particular Google and CloudFlare make use of this cipher in HTTPS connections even if it was not yet referred into IANA registered cipher list for TLS, since the cipher suite description for the protocol is today still in draft (Langley et al., 2015). Actually an approved RFC (Nir and Langley, 2015) for this AEAD cipher gives a generic implementation guide for IETF protocols.

An important feature used to promote Chacha20 is the possibility of being constant time by design. In facts, as highlighted in Section 5 of (Nir and Langley, 2015), all the operations involved in Chacha20 are additions, XORs and fixed rotations, which can be easily implemented in constant time in software. Cryptanalysis on Chacha20 can be found in (Aumasson et al., 2007), (Ishiguro, 2012), (Shi et al., 2013).

In Section 6.3 we report our benchmarks for AES128-GCM and Chacha20-Poly1305 AEAD ciphers.

4 THE SECURITY LEVEL OF A TLS CONNECTION

In this section we will discuss how to establish the security level of a TLS connection in relation to the negotiated cipher suite.

In (Barker et al., 2012) the authors provide a global view of the *security strength* and a key-size comparison between different public and private key algorithms binding them to the *bit of security* they give as summarized in Table 1. We can note that in order to provide 128 bit-security, the equivalent of AES-128, we have to choose at least a 3072-bit key for asymmetric ciphers like RSA or DH, and only a 256-bit key for ECC. Since in a TLS connection the master

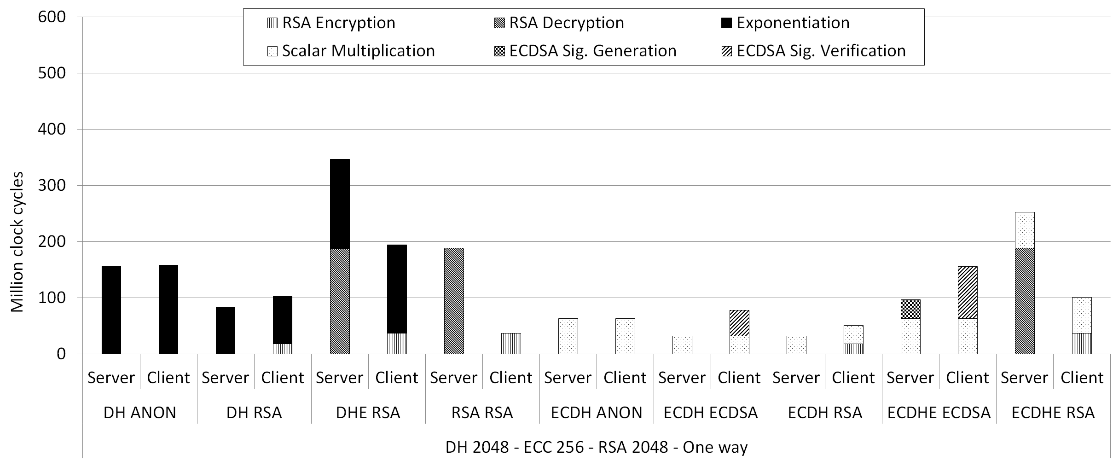

Figure 2: Benchmark of TLS 1.2 full handshake with one-way authentication.

Table 1: Comparable strengths.

Bits of security	Symmetric key algorithms	FFC/ IFC	ECC
80	2TDEA	1,024	160-223
112	3TDEA	2,048	224-255
128	AES-128	3,072	256-383
192	AES-192	7,680	384-511
256	AES-256	15,360	512+

secret used to generate the symmetric key is negotiated by employing public key algorithms, the security level of the connection is indeed upper bounded by the asymmetric cipher employed which can represent the bottle neck in terms of computational effort and memory requirements especially for IoT devices.

In the third part of the same NIST special publication (Barker et al., 2015), the authors describe the recommended key size for key establishment as RSA 2048, DH 2048 and ECDH P-256 or P-384. Note that for RSA 2048 and DH 2048 the equivalent symmetric key size is 112 bit, and in the case of ECC is 128 bit, thus employing a stronger symmetric cipher is not enough to provide more security to the connection, but also key sizes for asymmetric algorithms have to be incremented. This is the case, for example, of Chacha20-Poly1305 which is designed to provide 256 bit-security, but if the shared secret is negotiated with asymmetric algorithms like ECC 256, RSA 2048 or DH 2048, the security level of the connection is only 128 bit.

5 TLS 1.2: HANDSHAKE PERFORMANCE ANALYSIS

We started our work of analysis by performing bench marks for the TLS version 1.2. We build our testing environment by connecting our STM32F217IG[1] with an Intel Pentium M powered PC running Linux Ubuntu 12.04 LTS by an Ethernet cable. For cryptographic primitives and TLS implementation we employed the wolfSSL library [2] version 3.4.6 on the device and the OpenSSL library [3] version 1.0.2a on the PC. All the measurements reported in this article refers to the device side.

A previous similar work in which handshake performance are analyzed is (Koschuch et al., 2012), and this section can be an extension of Koschuch et al. paper, considering the new security standards discussed in Section 4. In fact it is important to note that in the choice of key sizes we followed the NIST recommendation reported in (Barker et al., 2015). In this respect, it is emphasised that due to the recent finding reported in (Adrian et al., 2015), the recommended key size for Diffie-Hellman key exchange algorithm has grown from 1024, which was used by millions of TLS, SSH, and VPN servers, to 2048. Figure 2 and 3 depict the clock cycles needed to complete an handshake when different cipher suites are negotiated, respectively for one way and mutual authentication schemes, while Table 6 in Appendix reports the values of memory usage. In case of one-way authentication, Figure 2 shows that the Diffie-Hellman exponentiation (see DH_RSA) is about 2.5 slower than the ECC (NIST P-256) scalar multiplication (e.g. ECDH_RSA), and when it comes to the ephemeral version, as in DH_ANON or DHE_RSA, an additional scalar multiplication is required for the key generation, increasing clock cycles of key exchange at a comparable level with RSA private key operations.

[1]www.st.com
[2]www.wolfssl.com
[3]www.openssl.org

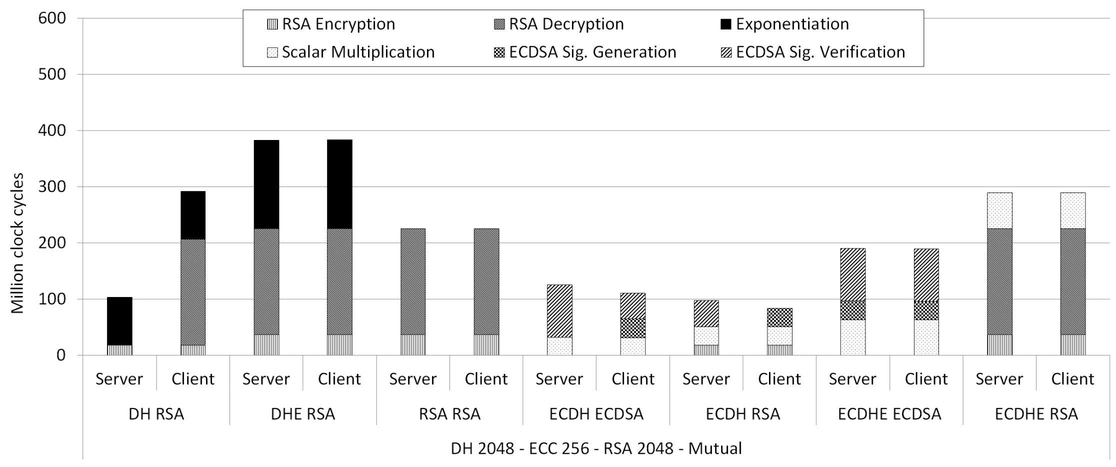

Figure 3: Benchmark of TLS 1.2 full handshake with mutual authentication.

Another aspect to be considered is the memory requirement: we observed that a Diffie-Hellman solution requires almost 1.5 times more RAM than the related ECC version. In particular, the DHE_RSA configuration demands the highest workload and memory occupation both for server and client, instead switching to Elliptic Curve induces a sensible reduction even with the ephemeral version.

When mutual authentication is employed (Figure 3), the client is required to perform private-key operations, and the clock cycles are mostly equivalent for client and server. In this case is even more evident the benefit of an Elliptic Curve solution. Comparing the ECDHE_ECDSA case with RSA_RSA one, we note that the former provide higher security by granting *Perfect Forward Secrecy* (PFS), and requires less computational effort and RAM for both client and server. Finally if we compare it with DHE_RSA, which also provides PFS, the ECC version requires half the time and less than 1.5 times the RAM. We also recall that, as discussed in Section 4, ECC P-256 provides 128 bit of security while DH 2048 and RSA 2048 only 112.

6 TLS 1.3 OVERHEAD ANALYSIS

Our aim is to evaluate the extent to which the novelties expected in TLS version 1.3 can impact on handshake and record protocols. For the evaluations presented in Subsections 6.2 and 6.3 we extended the wolfSSL library with our own cryptographic library by including the support for our new generation hardware accelerators.

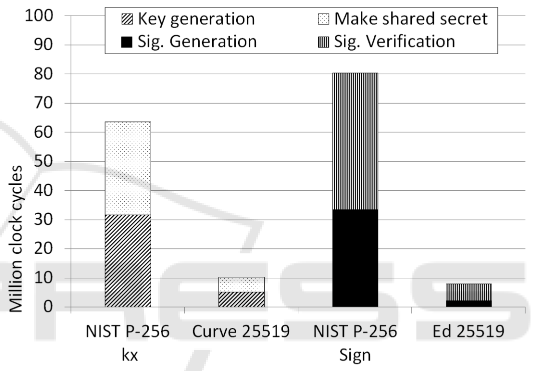

Figure 4: Elliptic curves benchmark.

6.1 On Ed25519 and Curve25519

Another important discussion among TLS experts concerns the adoption of new Elliptic Curves algorithms, in particular Curve25519 (Bernstein, 2006) for key exchange, and Ed25519 (Bernstein et al., 2012) for signature. The internet drafts that illustrate the use of such algorithms in TLS can be found at (Josefsson and Pegourie-Gonnard, 2015) and (Josefsson and Mavrogiannopoulos, 2015). These curves provide at least 128 bit security as NIST P-256, so we made some speed test and memory requirements comparison between them.

Figure 4 highlights that Curve25519 is 6 times faster than P-256 in a typical ephemeral key exchange which involves two exponentiations, one for generating the public key, and the other for shared key computation. Very impressive is the gap between NIST P-256 and Ed25519, in fact the latter is about 15 times faster in signature generation, and almost 8 times in signature verification. Another advantage of these new algorithms is the memory required at run-time, which is approximately a third for the key exchange, and about half for signature (Table 4 in the Appendix). If we take into account flash memory requirements like code size and constant data we note that NIST P-256 requires a little less than 20 KB for both key exchange and signature against the 70 KB needed by Curve25519 and Ed25519.

6.2 Handshake Encryption

As discussed in Section 2, the new TLS version 1.3 will introduce a new handshake flow in which all packets exchanged after the ServerKeyShare will be encrypted. This additional layer of encryption, in particular, allows the server to encrypt the response to any client's extensions which are not necessary to establish the Cipher Suite. The questions we want to answer are: a) how much this security improvement affects the time required for complete the handshake? b) Can this represent a problem for IoT devices?

Table 2: Overhead introduced for handshake encryption.

Certificate type	SW clock cycles	HW clock cycles
ECDSA	1,531,996	23,354
RSA	2,772,785	42,268

In order to evaluate the introduced overhead we simulated the encryption of a full TLS 1.2 handshake with AES128-GCM. Obviously the packets that require much workload are those who contain certificates, and, as reported in Table 2, the type of the certificates exchanged can impact on the computation. In fact the encryption of a certificate containing an RSA key requires almost double effort with respect to the one containing an ECC key. The Table furthermore illustrates the difference between clock cycles needed by software and hardware-accelerated implementations of the cipher.

Anyway even in the worst case (handshake involving an RSA-based certificate and no accelerators), the number of clock cycles is less than 3 Million. Comparing this value with the hundreds of millions clock cycles needed to carry out an handshake is clear that this additional layer does not introduce a sensible overhead and does not represent a bottle neck.

6.3 Application Data Encryption

The last evaluation is focused on application data encryption. Once the handshake is terminated, almost all the delay introduced by the TLS protocol towards a traditional non-secure socket is due to authenticated encryption. Hence the choice of symmetric cipher and authentication function is fundamental for both providing a security level commensurate with the asymmetric algorithm employed in the handshake, and ensuring throughput as high as possible.

In this section we present a performance analysis of AES128-GCM, Chacha20-Poly1305 and AES128-CBC-SHA which is the mandatory cipher for TLS 1.2 and relays on HMAC-SHA1 for the authentication, adopting a *MtE* strategy.

Table 3: Throughput comparison (Bytes/second) between Software and Hardware implementations of authenticated ciphers at 120MHz.

Algorithm	SW throughput	HW throughput
AES128-CBC+HMAC	578,726	6,142,506
AES128-GCM	304,216	8,757,487
Chacha20-Poly1305	771,962	NA

Table 3 reports the throughput of the ciphers at 120MHz encrypting packet of 15Kb. From the table it is evident that the software implementation of AES128 operated in GCM mode is almost 2 times slower than the CBC+HMAC version and 2.5 times than Chacha20-Poly1305. If we take a look at the hardware implementations the GCM mode of operation has the advantage that the authentication can be done in parallel with the encryption and the algorithm becomes faster than the CBC + HMAC variant.

With regard to memory requirements reported in Table 5 in the Appendix, we highlight that the tested software implementations requirements are: for AES128-GCM 16,4Kb of RAM and 14Kb of flash, for AES128-CBC 4.5Kb of RAM and 14.8Kb of flash, and for Chacha20-Poly1305 604 bytes of RAM and 3.3Kb of flash. Using the hardware accelerator makes possible to remove the software implementations of AES, HMAC and SHA1 and this results in improved code size for the AES-based ciphers, even less than the Chacha20-Poly1305 requirements. The reader should note that we are comparing algorithms of different key sizes. In fact, Chacha20 provides 256-bit security against the 128-bit of AES128-GCM and AES128-CBC.

7 DISCUSSION AND CONCLUSIONS

We can figure out two scenarios where IoT devices are employed in which different configuration can give their best. A first typical situation is video-surveillance in which high throughput is required and cryptography can really be a bottleneck. For this kind of application a device external-powered and equipped with cryptographic hardware accelerators is preferable, and a hardware-accelerated cipher like AES128-GCM should be employed. Otherwise in situations where battery life and economic factors are more important than throughput (e.g. wearables or environmental sensors), and amount of data to transmit are generally limited, it is reasonable to employ devices without criptographic hardware accelerators, and resources even more limited. Under these conditions a cipher like Chacha20-Poly1305 can be today a more suitable solution as it requires less resources both in space and clock cycles than the software AES-based implementations, allowing to save memory and reduce power consumption. Today AES-GCM and Chacha20-Poly1305 are the only two options as AEAD available in TLS. This might change in the coming years thanks to a competition called CAESAR[4]. In this competition there is an effort from researchers in evaluating new advanced AEAD ciphers.

In our analysis we focused on performances and memory requirements overhead introduced by the proposals in the TLS 1.3 draft, and we observed that even where the new security standards introduce an higher workload there is always an alternative that best matches IoT requirements.

The basic TLS 1.3 handshake flows introduce some new message types, as ServerConfiguration and EncryptedExtensions, and the encryption of sensitive handshake information, covering nearly two-thirds of the handshake messages. This solutions may burden low end devices. But on the side of cryptographic computations, the preference for AEAD ciphers and the adoption of new signature algorithms, as Curve25519 and Ed25519, may balance the increase in complexity, as shown by the simulation results of a commercial cryptographic library over an STM32 microcontroller.

Furthermore we provided use cases that let us conclude that the protocol still remain suitable for different classes of IoT devices.

REFERENCES

Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J. A., Heninger, N., Springall, D., Thomé, E., Valenta, L., et al. (2015). Imperfect forward secrecy: How diffie-hellman fails in practice.

Al Fardan, N. J. and Paterson, K. G. (2013). Lucky thirteen: Breaking the tls and dtls record protocols. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 526–540. IEEE.

Aumasson, J.-P., Fischer, S., Khazaei, S., Meier, W., and Rechberger, C. (2007). New features of latin dances: Analysis of salsa, chacha, and rumba. Cryptology ePrint Archive, Report 2007/472. http://eprint.iacr.org/.

Barker, E., Barker, W., Burr, W., Polk, W., Smid, M., Gallagher, P. D., et al. (2012). Recommendation for key management – part 1: General. *NIST special publication*, 800:57. Revision 3.

Barker, E., Burr, W., Jones, A., Polk, T., Rose, S., Smid, M., and Dang, Q. (2015). Recommendation for key management – part 3: Application-specific key management guidance. *NIST special publication*, 800:57. Revision 1.

Bellare, M., Kohno, T., and Namprempre, C. (2004). Breaking and provably repairing the ssh authenticated encryption scheme: A case study of the encode-then-encrypt-and-mac paradigm. *ACM Transactions on Information and System Security (TISSEC)*, 7(2):206–241.

Bellare, M. and Namprempre, C. (2000). Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology—ASIACRYPT 2000*, pages 531–545. Springer.

Bernstein, D. J. (2005). The poly1305-aes message-authentication code. In *Fast Software Encryption*, pages 32–49. Springer.

Bernstein, D. J. (2006). Curve25519: new diffie-hellman speed records. In *Public Key Cryptography-PKC 2006*, pages 207–228. Springer.

Bernstein, D. J. (2008). Chacha, a variant of salsa20. In *Workshop Record of SASC*, volume 8.

Bernstein, D. J., Duif, N., Lange, T., Schwabe, P., and Yang, B.-Y. (2012). High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89.

Borisov, N., Goldberg, I., and Wagner, D. (2001). Intercepting mobile communications: the insecurity of 802.11. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 180–189. ACM.

Degabriele, J. P. and Paterson, K. G. (2010). On the (in) security of ipsec in mac-then-encrypt configurations. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 493–504. ACM.

Dierks, T. and Rescorla, E. (2006). The transport layer security (tls) protocol version 1.1. RFC 4346, RFC Editor. http://www.rfc-editor.org/rfc/rfc4346.txt.

Dierks, T. and Rescorla, E. (2008). The transport layer security (tls) protocol version 1.2. RFC 5246, RFC Editor. http://www.rfc-editor.org/rfc/rfc5246.txt.

Dworkin, M. (2007). *Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC*. US Department of Commerce, National Institute of Standards and Technology.

Ford-Hutchinson, P. (2005). Securing ftp with tls. RFC 4217, RFC Editor. http://www.rfc-editor.org/rfc/rfc4217.txt.

[4]competitions.cr.yp.to/caesar.html

Gutmann, P. (2014). Encrypt-then-mac for transport layer security (tls) and datagram transport layer security (dtls). RFC 7366, RFC Editor. http://www.rfc-editor.org/rfc/rfc7366.txt.

Hoffman, P. (2002). Smtp service extension for secure smtp over transport layer security. RFC 3207, RFC Editor. http://www.rfc-editor.org/rfc/rfc3207.txt.

Ishiguro, T. (2012). Modified version of" latin dances revisited: New analytic results of salsa20 and chacha". *IACR Cryptology ePrint Archive*, 2012:65.

Josefsson, S. and Mavrogiannopoulos, N. (2015). Using eddsa with ed25519/ed448 in the internet x.509 public key infrastructure. Internet-Draft draft-josefsson-pkix-eddsa-04, IETF Secretariat. http://www.ietf.org/internet-drafts/draft-josefsson-pkix-eddsa-04.txt.

Josefsson, S. and Pegourie-Gonnard, M. (2015). Curve25519 and curve448 for transport layer security (tls). Internet-Draft draft-ietf-tls-curve25519-01, IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-tls-curve25519-00.txt.

Kent, S. and Atkinson, R. (1998). Ip encapsulating security payload (esp). RFC 2406, RFC Editor. http://www.rfc-editor.org/rfc/rfc2406.txt.

Koschuch, M., Hudler, M., and Krüger, M. (2012). The price of security: A detailed comparison of the tls handshake performance on embedded devices when using elliptic curve cryptography and rsa. In *e-Business and Telecommunications*, pages 71–83. Springer.

Krawczyk, H. (2001). The order of encryption and authentication for protecting communications (or: How secure is ssl?). In *Advances in Cryptology—CRYPTO 2001*, pages 310–331. Springer.

Langley, A., Chang, W.-T., Mavrogiannopoulos, N., Strombergson, J., and Josefsson, S. (2015). Chacha20-poly1305 cipher suites for transport layer security (tls). Internet-Draft draft-ietf-tls-chacha20-poly1305-03, IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-tls-chacha20-poly1305-03.txt.

McGrew, D. (2008). An interface and algorithms for authenticated encryption. RFC 5116, RFC Editor. http://www.rfc-editor.org/rfc/rfc5116.txt.

McGrew, D. and Viega, J. (2004). The galois/counter mode of operation (gcm). *Submission to NIST*. http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec. pdf.

Möller, B., Duong, T., and Kotowicz, K. (2014). This poodle bites: Exploiting the ssl 3.0 fallback.

Nir, Y. and Langley, A. (2015). Chacha20 and poly1305 for ietf protocols. RFC 7539, RFC Editor. http://www.rfc-editor.org/rfc/rfc7539.txt.

Rescorla, E. (2000). Http over tls. RFC 2818, RFC Editor. http://www.rfc-editor.org/rfc/rfc2818.txt.

Rescorla, E. (2015). The transport layer security (tls) protocol version 1.3. Internet-Draft draft-ietf-tls-tls13-10, IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-tls-tls13-10.txt.

Rogaway, P. (2002). Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 98–107. ACM.

Saint-Andre, P. (2011). Extensible messaging and presence protocol (xmpp): Core. RFC 6120, RFC Editor. http://www.rfc-editor.org/rfc/rfc6120.txt.

Salowey, J., Choudhury, A., and McGrew, D. (2008). Aes galois counter mode (gcm) cipher suites for tls. RFC 5288, RFC Editor. http://www.rfc-editor.org/rfc/rfc5288.txt.

Shi, Z., Zhang, B., Feng, D., and Wu, W. (2013). Improved key recovery attacks on reduced-round salsa20 and chacha. In *Proceedings of the 15th International Conference on Information Security and Cryptology*, ICISC'12, pages 337–351, Berlin, Heidelberg. Springer-Verlag.

Ylonen, T. and Lonvick, C. (2006). The secure shell (ssh) transport layer protocol. RFC 4253, RFC Editor. http://www.rfc-editor.org/rfc/rfc4253.txt.

APPENDIX

This Appendix reports the space requirements for elliptic curves primitives when performing signature and key exchange, AEAD ciphers software and - where is possible - hardware implementations, and negotiated cipher suite for completing a full TLS 1.2 handshake.

Table 4: Space requirements for elliptic curves expressed in bytes.

Algorithm	Stack	Heap	Total RAM	Code	Const. data	Total FLASH
NIST P-256 Key exchange	2,188	7,108	9,296	15,810	3,520	19,330
NIST P-256 Signature	2,052	9,316	11,368			
Curve25519	2,948	144	3,092	39,122	32,155	71,277
Ed25519	5,356	96	5,452			

Table 5: Space requirements for authenticated ciphers expressed in bytes.

Algorithm	Stack	Heap	Total RAM	Code	Const. data	Total FLASH
Chacha20-poly1305	604	0	604	3,300	52	3,352
AES128-GCM SW	4,804	11,644	16,448	3,288	10,846	14,134
AES128-CBC-HMAC-SHA1 SW	1,004	3,841	4,845	4,502	10,360	14,862
AES128-GCM HW	440	0	440	978	0	978
AES128-CBC-HMAC-SHA1 HW	712	0	712	2,180	10	2,190

Table 6: Space requirements for TLS 1.2 full handshake expressed in KBytes.

Cipher suite	Authentication	Client			Server		
		Stack	Heap	Total RAM	Stack	Heap	Total RAM
DH ANON	None	2.64	19.53	22.17	2.75	22.77	25.52
DH RSA	One way	3.91	21.27	25.18	2.75	22.79	25.54
	Mutual	3.91	22.47	26.38	3.96	23.13	27.09
DHE RSA	One way	3.91	21.48	25.39	2.75	22.78	25.53
	Mutual	3.92	24.37	28.29	3.96	24.64	28.6
RSA RSA	One way	3.91	13.55	17.46	2.75	18.55	21.3
	Mutual	3.91	21.16	25.07	3.96	20.41	24.37
ECDH ANON	None	2.53	10.37	12.80	2.69	12.36	15.05
ECDH ECDSA	One way	3.58	13.73	17.31	2.81	11.38	14.19
	Mutual	3.63	14.61	18.24	3.63	14.67	18.3
ECDH RSA	One way	3.91	13.41	17.32	2.75	11.86	14.61
	Mutual	3.92	14.62	18.54	3.96	15.69	19.65
ECDHE ECDSA	One way	3.58	13.79	17.37	2.75	12.06	14.81
	Mutual	3.58	14.67	18.25	3.63	14.98	18.61
ECDHE RSA	One way	3.91	13.66	17.57	2.75	19.1	21.85
	Mutual	3.91	21.27	25.18	3.96	19.1	23.06