Applied Cryptography - AttackHW Stage 4

Patrick Lee

1 Introduction

This specification attempts to re-design the firmware for the device $\mathcal{D}$ such that it can support some of the functionality required for TLS-based communications, and remain secure against implementation attacks, including side-channel and fault injection attacks. The scope of this design is limited to only support certain cryptographic functions required for TLS, with the assumption that the other operations involved in the protocol, such as the higher level steps involved in managing the handshake and record protocols, and the overarching networking as a whole, are handled by the host machine $\mathcal{H}$. It also assumes that only device $\mathcal{D}$ would be targeted in an implementation attack, and neither $\mathcal{H}$ nor the server it is communicating with, and that $\mathcal{D}$ is used as a 'clean slate', retaining no information about previous sessions.

2 Functionality and Implementation

This section outlines the cryptographic functions which should be implemented on $\mathcal{D}$, considering how they might be implemented in terms of the efficiency requirements and memory footprint constraints, as well as resistance to side-channel and fault injection attacks. It is assumed TLS1.3 **[1]** is being used in this context.

2.1 Key Agreement

'In line with the TLS1.3 specification, the Ephemeral Diffie-Hellman (DHE) **[2]** algorithm will be used for key agreement. As this relies on an ephemeral key, the implementation should be capable of generating a private key k_c some randomness supplied by $\mathcal{H}$. This key should be a prime from some finite field F, which has been approved as a safe-prime group as in **[2]**, and some generator g of this group should also be selected. The DHE implementation should then return a public key, $p_k = g^{k_c}$, to $\mathcal{H}$, which in response should supply the Server's public key, g^{k_s}, so that shared secret, $r = g^{k_c \cdot k_s}$, can be computed, completing the key agreement step. As the computation of the shared secret requires further input from the host (in the form of the received server public key), this should implemented within the key derivation function as in **2.2**

Any implementation of this will include a modular exponentiation step, both in forming the Host's public key and in computing the shared secret. This should employ Montgomery reduction and CRT to split the exponentiation into two smaller ones, to increase efficiency. Assuming a standard Binary-L2R exponentiation method is used, this could be exploited by an SPA attack or a timing attack to recover the Host's secret key, due to the data dependent sequence of squaring and multiplication operations giving a discernible variance in both power consumption and execution time. However, since performing this operation using k as an exponent will only happen twice in one session (the other being in Key Derivation **2.2**), the attacker would only be able to gain two power traces/execution timings of any relevance to k, significantly inhibiting the capability of attacks such as that described by Dhem et al. in **[3]** which used 50,000 execution timings.

With this said, protecting the private key is critical as recovering it would enable the attacker to de-crypt the entire session. Moreover, he small additional overhead required for a countermeasure here will not have a drastic effect on overall use as most of the TLS session occurs in the 'record' stage, after the handshake has happened.

To mitigate this vulnerability, a blinding method should be used to mask the private key k_c during the exponentiation operation. A random mask $m \in \mathbb{Z}$ should be selected and used to mask the key, $k' = k + m.\varphi(N)$. As $g^k \equiv g^{k'} \pmod N$. In-line with the CRT method, this should then be split into two exponentiations, $\sigma_p = g^{k_c \pmod{\Phi p.r}} \pmod{p.r}$ and $\sigma_q = g^{k_c \pmod{\Phi q.r}} \pmod{q.r}$, each of which should then be computed using Montgomery multiplication. σ_p can be subsequently compared with σ_q to ensure there is no discrepancy between values, providing a measure against a fault analysis attack **[4]**. σ_p and σ_q can then re-combined, and the public key is computed.

2.2 Key Derivation

The key derivation functionality should be an implementation of the HKDF function **[5]**, consisting of an extraction step, producing a pseudo-random key, and a subsequent expansion step, which should produce the keying material to be stored on $\mathcal{D}$ for the duration of the TLS session. Both of these make use of the HMAC mechanism, which in turn requires an implementation of an iterative hash function, such as those from the SHA family. The 'Extract' step will take the initial agreed key and a random salt as input and output a pseudo-random key. The 'Expand' step will then expand this into the required keying material. The HKDF scheme allows the option of skipping the 'extract' phase and moving straight to key expansion. This should **not** be skipped in this case, as the formed Diffie-Hellman value $g^{k_c.k_s}$ is not uniformly random. The 'extract' step is key in making it so. As the key derivation both operates and produces secret material, it is imperative that it is well protected against implementation attacks.

As in Key Agreement (**2.1**), the Key Derivation only needs to be executed once per TLS session. As such, an implementation can ensure that the derivation can only occur after a key generation/agreement execution. This would ensure that initial shared key can only be derived from on one occasion, before it is relinquished and a new session started, thereby ensuring an attacker is only able to gain a single side-channel reading for a given private key.

It has been shown in **[6]** that key derivation functions are vulnerable to timing attacks. Assuming the attacker can simulate a number of key derivations on the same architecture as $\mathcal{D}$, they could build a model relating different key derivation inputs to timings, and potentially use that to obtain some information about the keying material when observing execution time during operation of $\mathcal{D}$. Wei et al. described a countermeasure to this in **[7]**, which involves adding randomized delay into the algorithm, effectively masking the pattern of relationship between the execution time and parameter inputs involved. Whilst increasing the overhead of the operation slightly, the trade-off with improved security is favourable, particularly as this operation will only be performed once in a session, as in **2.1**. Therefore this should be implemented using as efficient a pseudo-random number generator as possible to determine the delay period.

2.3 End-point Authentication

During the handshake, the host $\mathcal{H}$ and the server will send each other signed certificates. Much of the security of this process depends upon the authenticity of the certificates and the reliability of the X.509 certification hierarchy **[8]**. To support certificate verification, basic RSA functionality should be implemented on $\mathcal{D}$ to compute the verification, sig^{pk_s} from the server's signature and public key, comparing it with the given certificate. Due to to the modular exponentiation, the implementation attacks outlined in **2.1** are also relevant here, although there is no secret information involved in the computation. A more realistic concern would be that of certificate forgery, and protections against this would be outside of the scope of this design.

2.4 Message Encryption and Authentication

Both the encryption and the authentication of messages sent over TLS can be handled with an implementation of AES-GCM **[9]**, using the AES-128 functionality implemented in **stage 3** (with the addition of decryption functionality supported by an adaptation of the existing countermeasures) as the underlying block cipher. As such, the encryption and decryption can keep the encryption key protected from CPA attacks by using some combination of masking and temporal dis-alignment as outlined **stage 3**. As this functionality will be used frequently throughout the TLS session, some trade-off may have to be made between keeping the overhead of the counter-measures low and the required security.

This implementation should require a plaintext, some additional authenticated data (such as protocol information), which is not to be encrypted but should be authenticated, to be sent from $\mathcal{H}$, as well as an IV/nonce. It should then return a cipher text containing the encrypted plaintext, an authentication tag and padding field. When decrypting, it should return the inverse of this, or an error if authentication fails. It is critical that the IV or nonce used in the generation of the authentication tag is not repeated during a TLS session.

The authentication functionality in AES-GCM requires some implementation of GHASH, making use of Galois field multiplication **[9]** for authentication. These are somewhat expensive operations and are often sped up by using lookup tables **[11]**. A balance would have to be found between the limited memory offered by $\mathcal{D}$ and the desired speed of authentication. It should be ensured that the implementation is resistant to attacks which attempt to recover the hash key H used in the authentication, as doing so would allow the attacker to stage a forgery attack. If the underlying AES implementation is secure against CPA, it is reasonable to assume the attacker would not be able to recover this hash key from analysing its initial computation from a 128-bit string of zeros, $H = E_k(0^{128})$

Fahd et al. showed in **[10]** that an attacker could leverage power analysis to model hamming weight of look-up table access operand during the multiplication step of GCM. Blinding H with $E_k(Y_0)$ and data dependent rotations based upon 7 right most bits of $H \oplus E_k(Y_0)$ was suggested as counter-measure. This blinding should be implemented before any operations using H take place. It should also be noted that standardized GCM structure is tolerant against fault injections **[10]**. This counter-measure retains this resistance as a single-bit flip will produce an erroneous H that will propagate and disturb subsequent H generation for next blocks, generating an erroneous tag which would be rejected upon decryption. Hence implementing this counter-measure will ensure resistance to both power analysis and fault-injection attacks.

A variant of this attack proposed in **[11]** exploits the timing variability of data loads from memory. This variability is due to the fact that all modern microprocessors with caches use a hierarchy of caches to reduce load latency. However, in this application, $\mathcal{D}$ does not have an in-built data or instruction cache **[12]**, so this side-channel attack can be ignored.

3 API (Communication between $\mathcal{H}$ and $\mathcal{D}$)

Overall, the protocol for communication between the host machine, $\mathcal{H}$ and the device $\mathcal{D}$, is simply an extension of that described in Section C of **[13]**, using the same command-based protocol and octet string format. The commands that correspond with this design specification are outlined below.

- The suite command exposes the cipher suite that $\mathcal{D}$ is configured to use..
  - $\mathcal{H}$ sends command $cmd = 00_{(16)}$ to $\mathcal{D}$
  - $\mathcal{D}$ sends the cipher suite (a sequence of 2 bytes) to $\mathcal{H}$

- The key generation command generates a DHE key pair for the purposes of key agreement, retaining the private key k_c and returning the public key pk_c.
  - $\mathcal{H}$ sends command $cmd = 01_{(16)}$ to $\mathcal{D}$
  - $\mathcal{D}$ sends the public key (a sequence of N bytes) to $\mathcal{H}$
- The key derivation command computes the shared master secret, from the previously generated private key k_c and the given server public key pk_s and derives and retains the required key material from this.
  - $\mathcal{H}$ sends command $cmd = 02_{(16)}$ to $\mathcal{D}$
  - $\mathcal{H}$ sends the server's public key (a sequence of N bytes) to $\mathcal{D}$
  - $\mathcal{H}$ sends a random salt value (a sequence of N bytes) to $\mathcal{D}$
  - $\mathcal{D}$ sends a flag (single byte) to $\mathcal{H}$ indicating if the derivation was successful or not. For instance, if the operation could not complete because a key pair had not yet been generated
- The endpoint verification command verifies a digital certificate with the endpoint's public key and a given signature
  - $\mathcal{H}$ sends command $cmd = 03_{(16)}$ to $\mathcal{D}$
  - $\mathcal{H}$ sends the certificate (a sequence of N bytes) to $\mathcal{D}$
  - $\mathcal{H}$ sends the endpoint's public key (a sequence of N bytes) to $\mathcal{D}$
  - $\mathcal{H}$ sends the given signature (a sequence of N bytes) to $\mathcal{D}$
  - $\mathcal{D}$ sends a flag to indicate successful authentication of the endpoint(a single byte) to $\mathcal{H}$
- The authenticated encryption command computes a cipher-text and authentication tag using the previously computed shared secret and a user supplied plaintext, additional data and IV
  - $\mathcal{H}$ sends command $cmd = 04_{(16)}$ to $\mathcal{D}$
  - $\mathcal{H}$ sends the plain-text (a sequence of BLK bytes) to $\mathcal{D}$
  - $\mathcal{H}$ sends the additional authenticated data (a sequence of BLK bytes) to $\mathcal{D}$
  - $\mathcal{H}$ sends a random IV to (a sequence of BLK bytes) to $\mathcal{D}$
  - $\mathcal{D}$ sends a ciphertext (a sequence of BLK bytes) to $\mathcal{H}$
  - $\mathcal{D}$ sends an authentication tag (a sequence of BT bytes) to $\mathcal{H}$
- The authenticated decryption command computes a plaintext from a ciphertext and authentication tag, generating an error code if authentication was unsucessfull
  - $\mathcal{H}$ sends command $cmd = 05_{(16)}$ to $\mathcal{D}$
  - $\mathcal{H}$ sends the cipher-text (a sequence of BLK bytes) to $\mathcal{D}$
  - $\mathcal{H}$ sends the authentication tag (a sequence of T bytes) to $\mathcal{D}$
  - $\mathcal{H}$ sends the additional authenticated data (a sequence of BLK bytes) to $\mathcal{D}$
  - $\mathcal{H}$ sends a random IV to (a sequence of BLK bytes) to $\mathcal{D}$
  - $\mathcal{D}$ sends a flag to indicate successful authentication (a single byte) to $\mathcal{H}$
  - $\mathcal{D}$ sends the plain-text **if** authentication is sucessful to $\mathcal{H}$

References

[1] E. Rescorla The Transport Layer Security (TLS) protocol version 1.3 Internet Engineering Task Force (IETF) Request for Comments (RFC) 8446. 2018. `http://tools.ietf.org/html/rfc1321`

[2] Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography. National Institute of Standards and Technology (NIST) Special Publication 800-56A. 2007. url: `http://csrc.nist.gov`

[3] Dhem, JF., Koeune, F., Leroux, PA., Mestré, P., Quisquater, JJ., Willems, JL. (2000). A Practical Implementation of the Timing Attack. In: Quisquater, JJ., Schneier, B. (eds) Smart Card Research and Applications. CARDIS 1998. Lecture Notes in Computer Science, vol 1820. Springer, Berlin, Heidelberg. url: https://doi.org/10.1007/10721064_15

[4] A. Shamir. Method and Apparatus for protecting public key schemes from timing and fault attacks. U.S. Patent 5,991,415. url: http://www.google.com/patents/US5991415

[5] H. Krawczyk HMAC-based Extract-and-Expand Key Derivation Function (HKDF). Internet Engineering Task Force (IETF) Request for Comments (RFC) 5869. 2010. url: http://tools.ietf.org/html/rfc5869

[6] Chuah, C.W., Koh, W.W. (2017). Timing Side Channel Attack on Key Derivation Functions. In: Kim, K., Joukov, N. (eds) Information Science and Applications 2017. ICISA 2017. Lecture Notes in Electrical Engineering, vol 424. Springer, Singapore. url: https://doi.org/10.1007/978-981-10-4154-9_31

[7] Wei, C.C.Z., Wen, C.C., Leakage Resilient of Timing Side Channel Attack for Key Exchange Security Model. International Journal of Innovative Technology and Exploring Engineering (IJITEE) ISSN: 2278-3075, Volume-8, Issue-8S, June 2019. url: https://www.ijitee.org/wpcontent/uploads/papers/v8i8s/H10430688S19.pdf

[8] D. Cooper et al. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Internet Engineering Task Force (IETF) Request for Comments (RFC) 5280. 2008. url: http://tools.ietf.org/html/rfc5280

[9] Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. National Institute of Standards and Technology (NIST) Special Publication 800-38D. 2007. url: http://csrc.nist.gov

[10] Shah Fahd, Mehreen Afzal, Haider Abbas, Waseem Iqbal, Salman Waheed. Correlation power analysis of modes of encryption in AES and its countermeasures. Future Generation Computer Systems, Volume 83, 2018, Pages 496-509, ISSN 0167-739X, url: https://doi.org/10.1016/j.future.2017.06.004.

[11] Käsper, E., Schwabe, P. (2009). Faster and Timing-Attack Resistant AES-GCM. In: Clavier, C., Gaj, K. (eds) Cryptographic Hardware and Embedded Systems - CHES 2009. CHES 2009. Lecture Notes in Computer Science, vol 5747. Springer, Berlin, Heidelberg. url: https://doi.org/10.1007/978-3-642-04138-9_1

[12] ARM Cortex-M https://en.wikipedia.org/w/index.php?title=ARM_Cortex-M

[13] Applied Cryptology (COMS30048) Assessed coursework assignment AttackHW url: http://assets.phoo.org/COMS30048_2022_TB-2/csdsp/cw/AttackHW/question.pdf