Table of Contents

Transport Layer Security Protocol For Internet Of Things Extended Abstract

Illya Gerasymchuk
illya@iluxonchik.me

Instituto Superior Tecnico
Supervisors: Ricardo Chaves, Aleksandar Ilic

Abstract. Transport Layer Security (TLS) is one of the most used communication security protocols in the world. It comes with many configurations. Each configuration offers a set o security services, which has an implication on the security level and computational cost. Not all of those configurations can be used with the resource constrained Internet Of Things (IoT) devices, due to the high computational and memory demands. Most of the existing work focuses on Datagram TLS (DTLS) and cannot be easily integrated with existing deployments. Existing work fails to evaluate the cost of various TLS configurations and its security services. This work focuses on cost analysis of the security services of the TLS protocol. We evaluate the number of CPU cycles used by the TLS configurations and by each individual security service. Software developers can use this information to make security/cost trade-offs based on the environment needs and limitations.

Keywords: TLS, DTLS, SSL, IoT, cryptography, protocol, lightweight cryptography

1 Introduction

In recent years there has been a sharp increase in the number of IoT devices and this trend is expected to continue. The IoT is a network of interconnected devices, which exchange data with one another over the internet. While they are many types of IoT devices, all of them are restricted: they have limited memory, processing power and available energy. Examples of IoT devices include temperature sensors, smart light bulb and physical activity trackers.

While inter-device communication has numerous benefits, it is important to ensure the security of that communication. For example, when you log in to your online banking account, you do not want others to be able to see your password, as this may lead to the compromise of your account. Having your account compromised means that a malicious entity might steal your money. Similarly, when you are transferring funds via online banking, you want the contents of that operation to be invisible to an observer, for privacy reasons. It is also desirable that no party is able to tamper with the transmitted data en transit, as it may lead to undesired consequences, such as the transfer of a larger amount than you intended. Proper communications security allows those goals to be achieved.

TLS is one of the most used protocols for communication security. It powers numerous technologies, such as Hypertext Transfer Protocol Secure (HTTPS). TLS offers the security services of authentication, confidentiality, privacy, integrity, replay protection and perfect forward secrecy. It is not a requirement to use all of those services for every TLS

connection. The protocol is similar to a framework, in the sense that you can enable individual security services on a per-connection basis. For example, when you are downloading software updates, data confidentiality is probably not a concern, data authenticity and integrity, however, is. In TLS, it is possible for a connection to only offer authenticity and integrity, without offering confidentiality. Foregoing unnecessary services will lead to a smaller resource usage, which in turn leads to smaller execution time and power usage. This is especially important in the context of IoT, due to the constrained nature of the devices.

There are numerous IoT devices, each one with different hardware capabilities and security requirements. For example, some IoT devices have the capability of using public key cryptography, while for others symmetric cryptography is the only option In some cases, the communicating devices require data authenticity, confidentiality and integrity (e.g. when logging in into a device), while in others data authenticity and integrity is enough (e.g. when transferring updates).

TLS was not designed for the constrained environment of IoT. Despite that, it is a malleable protocol and can be configured to one's needs. In essence, it is a combination of various security algorithms that together form a protocol for communication security. If configured properly, it is possible to use it in the context of IoT. Existing work does not address the computational cost evaluation of the various security services offered by TLS individually. An example of a computational is the number of CPU cycles used, time taken or power used. Thus, software developers who want to use TLS to provide connection security do not have data that can assist them in making the security/resource usage trade-offs

The goal of this work is to evaluate the cost of each security service of the TLS protocol. This will assist software developers to make security/resource usage trade-off judgments, according to their needs and limitations. For this reason, this work targeted towards developers who wish to add communication security to their applications in the IoT environment.

In the process of the work on this dissertation, we have made several contributions to the TLS 1.3 specification, and were formally recognized as contributors[1]. Our name can be found in the document specifying TLS 1.3[2]. We have also found a security issue within the TLS implementation of the *mbedTLS* library. We reported it and it has been assigned a *CVE* with the id *CVE-2018-1000520*[3]. It is an authentication problem, where certificates signed with an incorrect algorithm were accepted in some cases. More specifically, *ECDH(E)-RSA* ciphersuites allowed Elliptic Curve Digital Signature Algorithm (ECDSA)-signed certificates, when only Rivest-Shamir-Adleman (RSA)-signed ones should be. We also found a bug in *mbedTLS*'s test suite related to the use of deprecated *SHA-1*-signed certificates and submitted a code fix to it [4][5].

2 Motivation

The majority of existing work on (D)TLS ((D)TLS) optimization proposes a solution that is either tied to a specific protocol, such as Constrained Application Protocol (CoAP), or requires an introduction of a third-party entity, such as the trust anchor in the case of the S3K system[6] or even both. This has two issues. First, a protocol-specific solution cannot be easily used in an environment where (D)TLS is not used with that protocol. Second, the requirement of a third-party introduces additional cost and complexity, which will be a big resistance factor in adopting the technology. This is specially true for developers

working on personal projects or projects for small businesses, leaving the communications insecure in the worse case scenario. Therefore a solution that is protocol independent and fully compatible with the (D)TLS standard and existing infrastructure is desired.

Another issue with the existing literature is that it almost exclusively focuses on DTLS optimization and not all of it can be applied to TLS. Herein we want to further explore TLS optimization. There is clearly a need for that, specially with CoAP over TCP and TLS standard being currently developed. The mentioned standard does not explore any TLS optimizations, and since any IoT device using it in the future would benefit from them, this is an important area to explore.

(D)TLS is a complex protocol with numerous possible configurations. Each configuration provides different set security services and a different security level. This has a direct impact on the resource usage. Thus, the cost of a (D)TLS connection can be lowered, by using an appropriate configuration. Typically, this involves making security/cost trade-offs. Optimizing the connection cost in this way meets our goals of being protocol independent, fully compatible with existing infrastructure and targeting TLS optimization specifically.

The objective of this work is to provide a means of assisting application developers who wish to include secure communications in their applications to make security level/resource usage trade-offs, according to the environment's needs and limitations. In order to achieve this goal, the costs of each individual security service will be evaluated. With this information, the programer will be able to choose a configuration that meets his security requirements and device constraints. If the limitations of the device's hardware do not allow to meet the requirements, the programer may decide on an alternative configuration, possibly with a loss of some security services and a lower security level, or forgo using (D)TLS altogether.

3 Methodology

In our work, we evaluated the *mbedTLS's* implementation of the TLS protocol, thus obtained metrics reflect the algorithm's implementations used within the library. The metric that we evaluated is the number of CPU cycles executed. To be more precise, an estimate of that value. In order to estimate the number of executed cycles, we used *valgrind*, more specifically its *callgrind* tool. *valgrind* runs the application on a synthetic CPU. While running the code in that synthetic environment, it is able to insert instructions to perform profiling and debugging. In essence, *valgrind* is a virtual machine, using just-in-time (JIT) compilation techniques, such as dynamic recompilation.

Among other metrics, *callgrind* collects the number of executed instructions, L1/L2 caches misses (the caches are simulated), and branch prediction misses. The metrics collected by *callgrind* can then be loaded into *kcachegrind* to visualize and analyze the performance results. One of such results is the estimate of the number of executed CPU cycles. *callgrind* and *kcachegrind* are widely in conjunction for performance analysis and optimization of programs. In order to count the number of executed CPU cycles, we used the following formula (derived from the one used by *kcachegrind*): $CEst = Ir + 10 * Bm + 10 * L1m + 100 * L2m$, where Ir is the number of instruction fetches, Bm is the number of mispredicted branches and $L1m$ and $L2m$ are the number L1 and L2 cache misses, respectively.

In order to estimate the number of CPU cycles, we developed automated tooling that collected and analyzed the metrics output by *callgrind*. All of the evaluations were performed in our local *Intel(R) Core(TM) i7-4700HQ* machine.

4 Evaluation

The (D)TLS protocol consists of two sub-protocols: the Handshake protocol and the Record protocol. During the Handshake protocol, the peers authenticate one to another, agree on the data encryption and integrity algorithms and establish the shared keys. It offers the security services of confidentiality and integrity. During the Record protocol the peers exchange the data securely, using the algorithms and keys negotiated during the Handshake protocol. Those algorithms support the security services provided by (D)TLS.

The core of the Record protocol is the use of a symmetric encryption algorithm (*e.g* AES), used to provide confidentiality and an Hash-Based Messaage Authentication Code (HMAC) algorithm, used to provide data integrity and data origin authentication. The performance of symmetric encryption algorithms and hash functions has been studied in detail by existing work [7] [8]. There is an approximately linear relation between the amount of data encrypted and the the cost.

The previous paragraph does not apply to the Handshake protocol. In this part, there is more variety in the computational cost outcomes. There are a few reasons for that. First, there are significantly more possible key sizes (unlimited, in theory), when compared to symmetric encryption algorithms. Second, various algorithms, in various combinations can be used to provide authentication and PFS. Third, the use of asymmetric cryptography leads to asymmetric costs (*i.e.* distinct costs) for the client and the server. The costs of asymmetric encryption algorithms vary greatly depending if the public or the private key is used.

It is clear that the Handshake protocol is significantly more complex, with more possible cost variations. Furthermore, existing work neither evaluated the costs of individual security services of TLS, nor their various combinations. For this reason, our work is concentrated around the Handshake protocol.

We evaluated the costs for various security services, defined in table 1. We analyzed for the most common authentication scenario only: only the server authenticates to the client. As a consequence, the costs of authentication are different for the client and the server.

	low	**normal**	**high**	**very high**
Symmetric Key Size (bits)	128	128	192[1]	256
RSA/DH/DSA Key Size (bits)	1024	2048	4092	8192
ECC Key Size (bits)	163 [2]	233 [3]	317[4]	420[5]
HMAC	SHA-256	SHA-256	SHA-384	SHA-512 [6]

Table 1: Security levels used in evaluation

[1] The closest value available in mbedTLS 2.7.0 (rounded up) is 256 bit
[2] The closest value available in mbedTLS 2.7.0 (rounded up) is 224 bit
[3] The closest value available in mbedTLS 2.7.0 (rounded up)is 256 bit
[4] The closest value available in mbedTLS 2.7.0 (rounded up) is 384 bit
[5] The closest value available in mbedTLS 2.7.0 (rounded up) is 512 bit
[6] The strongest hash function available in mbedTLS is SHA-384

4.1 Authentication Cost Analysis

In TLS there are two ways of doing authentication: either by using a Pre-Shared Key (PSK) or by using asymmetric cryptography.

If only PSK is used for authentication (*PSK* key exchange), the cost of authentication is practically 0. In this case, the peers perform the simplest handshake available in *mbedTLS 2.7.0*. The cost of a handshake that uses the *PSK* can be considered as the overhead of establishing the TLS connection.

If asymmetric cryptography is used for authentication, there are two choices of algorithms: RSA and ECDSA. Each one of them has advantages and disadvantages, depending on the scenario. RSA is more efficient at performing public key operations, *i.e.* verifying the signature. ECDSA is more efficient at performing private key operations, *i.e.* making the signature. Figure 1 shows the costs of both, RSA's and ECDSA's operations. The data is presented in logarithmic scale. The *Total* cost is the sum of the costs of the signature creation and verification operations. Since for RSA most of the *Total* cost comes from the signature creation operation, those two lines overlap in the graph.

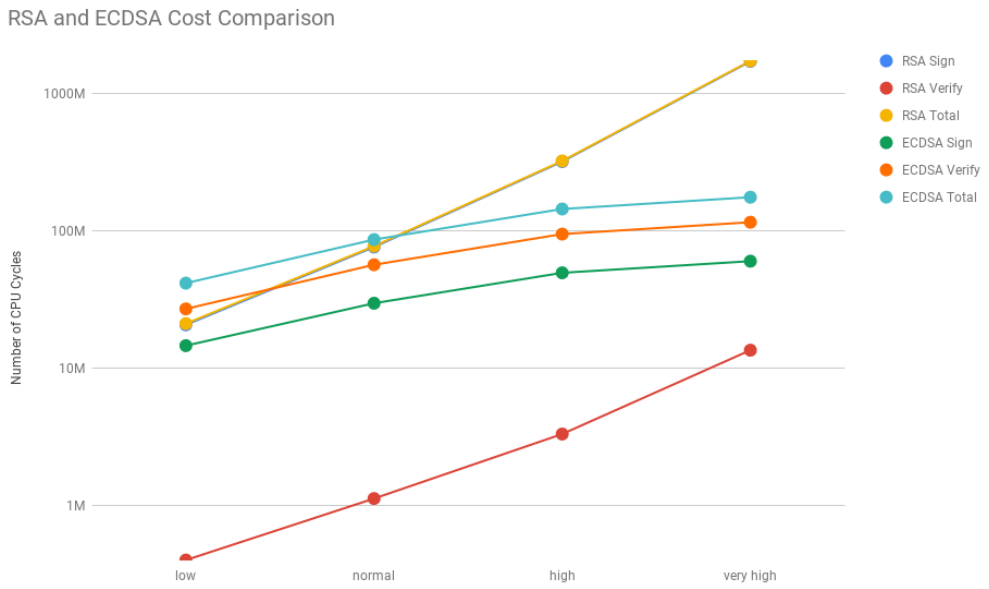

Fig. 1: RSA and ECDSA cost comparison

By analyzing Figure 1, we can contrast both of the algorithms. First, while RSA is always less costly at signature verification, ECDSA is always less costly at signature creation, Second, while RSA's cost increase is exponential,ECDSA's cost increase is logarithmic. And finally, while the total cost of RSA is smaller for the *low* and *normal* security levels, the total cost of ECDSA is smaller for the *high* and *very high* security levels.

The properties of both algorithms have a direct impact on the TLS handshake costs. Table 2 shows authentication costs for each ciphersuite for the client. Table 3 shows the

same information for the server. Each row specifies a key exchange method and each column the security level.

	low	**normal**	**high**	**very high**
PSK	0	0	0	0
RSA	654077	2622235	5285561	16398134
RSA-PSK	654077	2622235	5285561	16398134
ECDH-RSA	1117868	1117868	1117868	1117868
ECDH-ECDSA	56260702	56260702	56260702	56260702
ECDHE-PSK	0	0	0	0
ECDHE-RSA	1516452	2235736	4413164	14554596
ECDHE-ECDSA	83099975	112521404	150337852	171004723
DHE-PSK	0	0	0	0
DHE-RSA	1516452	2235736	4413164	14554596

Table 2: Client authentication costs for all ciphersuites and security levels

	low	**normal**	**high**	**very high**
PSK	0	0	0	0
RSA	20362831	75129504	314975365	1691976601
RSA-PSK	20362831	75129504	314975365	1691976601
ECDH-RSA	0	0	0	0
ECDH-ECDSA	0	0	0	0
ECDHE-PSK	0	0	0	0
ECDHE-RSA	20559190	75738802	317087210	1700652764
ECDHE-ECDSA	14497591	29512991	49150396	59732056
DHE-PSK	0	0	0	0
DHE-RSA	20559190	75738802	317087210	1700652764

Table 3: Server authentication costs for all ciphersuites and security levels

4.2 PFS Cost Analysis

In TLS there are two ways of achieving PFS: either by using the Diffie-Hellman (DH) algorithm or its Elliptic Curve Cryptography (ECC) counterpart Elliptic Curve Diffie-Hellman (ECDH).

Figure 2 compares the costs of ECDH and DH. If the **low** security level is being used, DH is the least costly choice, if the **normal** or any security level above is being used, ECDH is. Moreover, we can clearly see that for each algorithm, the costs of their operations is very similar, since we have overlapping lines. The logarithmic and exponential properties of ECDH and DH, respectively, are also visible by shape of the lines. Since we are using logarithmic scale for the y axis, the exponential cost growth of DH manifests in the shape of a line.

Even though the *ECDH* key exchange does not offer PFS, it is still closely related to the *ECDHE*, since both use the ECDH algorithm. The additional costs for *ECDH* ciphersuites

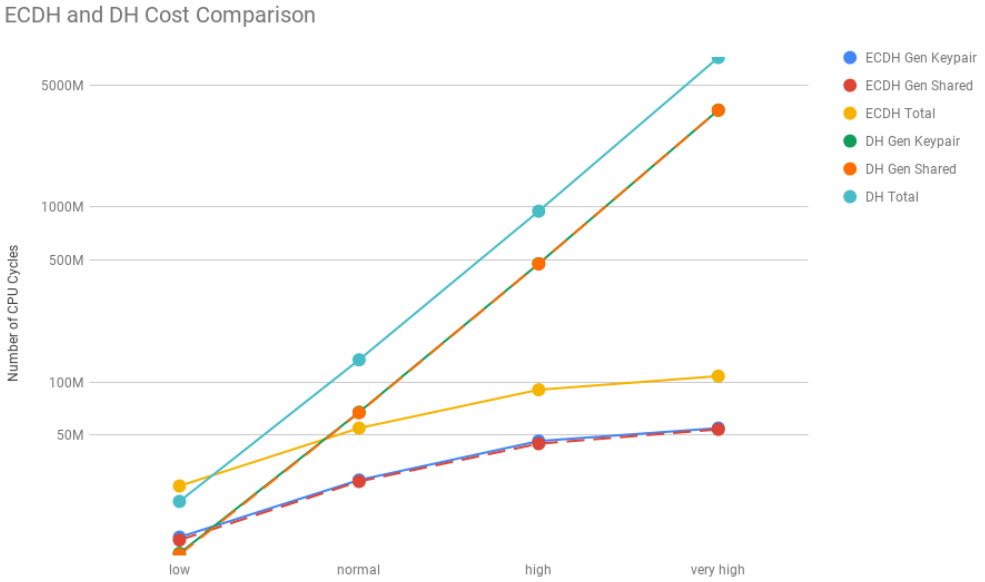

Fig. 2: ECDH and DH cost comparison (logarithmic scale)

are significant, and in our evaluated scenario where only the server authenticates to the client, there is no distinction in costs between *ECDHE* and *ECDH* ciphersuites for the client. Tables 4 and 5 show the cost of PFS and the *ECDH* key exchanges for the client and the server, respectively.

	low	**normal**	**high**	**very high**
PSK	0	0	0	0
RSA-PSK	0	0	0	0
RSA	0	0	0	0
ECDH-RSA	25405195	54442524	90231688	107981294
ECDH-ECDSA	25405195	54442524	90231688	107981294
ECDHE-PSK	25405195	54442524	90231688	107981294
ECDHE-RSA	25405195	54442524	90231688	107981294
ECDHE-ECDSA	25405195	54442524	90231688	107981294
DHE-PSK	20734678	133849027	948573054	7178955325
DHE-RSA	25405195	133849027	948573054	7178955325

Table 4: PFS and *ECDH* key exchange for the client

4.3 Handshake Cost Analysis

Having analysed the cost of the security services of authentication and PFS, we are now ready to analyze the cost of the handshake. Tables 6 and 7 show the average handshake

	low	normal	high	very high
PSK	0	0	0	0
RSA-PSK	0	0	0	0
RSA	0	0	0	0
ECDH-RSA	12462677	26958612	44331330	53531554
ECDH-ECDSA	12462677	26958612	44331330	53531554
ECDHE-PSK	25405195	54442524	90231688	107981294
ECDHE-RSA	25405195	54442524	90231688	107981294
ECDHE-ECDSA	25405195	54442524	90231688	107981294
DHE-PSK	20734678	133849027	948573054	7178955325
DHE-RSA	20734678	133849027	948573054	7178955325

Table 5: PFS and *ECDH* key exchange for the server

cost for each one of the key exchange methods, for the client and server, respectively. We can use the following formula to compute the cost of the TLS Handshake cost for a key exchange method and security level: $HandshakeCost = TLSOverhead + AuthCost + PFSCost + AdditionalCosts$. The *TLS Overhead* is the cost of the *PSK* key exchange and can be obtained from table 6 for the client and from table 7 for the server. *Auth Cost* is the cost of authentication and can be obtained from 2 for the client and from table 3 for the server. *PFS* cost is the cost of PFS and can be obtained from table 4 for the client and from table 5 for the server. *Additional Costs* come from the *Certificate* message. This message is omitted entirely in the *PSK* key exchange. In ciphersuites that use asymmetric authentication, the client has to read the *Certificate* message from the record layer, parse the *der* encoded certificate into internal fields and check the validity of its fields, such as the not valid before/after dates. The server has to write the *Certificate* message to the record layer. This cost for the client is approximately 622615.

As an example, we will compute the client's Handshake cost for the **high** security level with *ECHDE-RSA* First, we will need to get the *TLSOverhead* value. We can obtain in from the entry located at the *ECDHE-RSA* row and *high* column in table 6: 1355005. *AuthCost* can be obtained from the *ECDHE-RSA* row and *high* column in table 2: 4413164. Similarly *PFSCost* can be obtained from table 4, where the *ECDHE-RSA* row and *high* column intersect: 90231688. The *AdditionalCosts* value is 622615. All that's left now is insert those values in the formula: $HandshakeCost = TLSOverhead + AuthCost + PFSCost + AdditionalCosts = 1355005 + 4413164 + 90231688 + 622615 = 96622472$. As we can see, the cost computed using the formula is very close to the actual Handshake cost when evaluated as a whole, which can be found in in table 6. The difference of 108429 ($96730901 - 96607399$) CPU cycles comes from other additional costs, such as reading and writing slightly larger messages.

4.4 Confidentiality Cost Analysis

Our evaluation of the encryption algorithms in *mbedTLS 2.7.0* showed that *3DES* is the most costly one of all, while Authenticated Encryption With Associated Data (AEAD) encryption algorithms (*AES* with *GCM* and *CCM* modes) are the least costly block cipher algorithms. *CAMELLIA* algorithms are more costly than their *AES* counterparts.

In order to answer the question of how much data it is needed to send (thus encrypt) in order to equate the cost of the handshake, we derived the cost formula for the *AES-128-GCM* (suitable **low** and **normal** security levels) and *AES-256-GCM* (suitable **high**

	low	**normal**	**high**	**very high**
PSK	1354543	1353865	1355005	1354829
RSA-PSK	3536763	4543238	7293473	18578392
RSA	3556430	4558935	7309587	18608492
ECDH-RSA	28433250	57412269	93325335	111532116
ECDH-ECDSA	83731926	112585306	148415289	166815336
ECDHE-PSK	26819362	55805138	91713096	109203285
ECDHE-RSA	28862986	58608367	96730901	124773055
ECDHE-ECDSA	110524836	168954007	242501399	280626510
DHE-PSK	22189619	135442518	950212732	7181764421
DHE-RSA	24250886	138197524	955840114	7196088947

Table 6: Handshake costs for the client

	low	**normal**	**high**	**very high**
PSK	1380956	1382849	1381497	1382002
RSA-PSK	22515609	77368868	317389619	1694178055
RSA	22456180	77260738	317196961	1694747500
ECDH-RSA	14534087	29154666	46480720	55636527
ECDH-ECDSA	14521154	29094320	46390603	55927921
ECDHE-PSK	26869449	56020895	91773474	109500050
ECDHE-RSA	48164890	132433386	409652623	1810563620
ECDHE-ECDSA	41984222	86096716	141506181	169996896
DHE-PSK	22308029	135535409	950691653	7181243788
DHE-RSA	43602094	212251542	1268235354	8883866054

Table 7: Handshake costs for the server

and **very high** security levels) algorithms. We selected those algorithms because they were among the cheapest ones to provide the required security level and are preferred by browsers, such as *Google Chrome 67*. The cost of encryption and decryption for those algorithms are similar[9].

The formulas for the cost of encryption with *AES-128-GCM* and *AES-256-GCM*, respectively are: $NumCC = 104 * NumBytes + 22680$ and $NumCC = 105 * NumBytes + 22740$. $NumCC$ is the number of CPU Cycles and $NumBytes$ is the number of bytes encrypted. As an example, let us compute how many bytes must be sent for the cost of encryption to equate the cost of an *ECDHE-ECDSA* handshake at the *normal* security level. First, we obtain the number of CPU cycles used to perform the handshake from table 6: 168954007. After that we simply replace $NumCC$ with that value in the formula and solve for $NumBytes$: $168954007 = 104 * NumBytes + 22680 \Leftrightarrow NumBytes \approx 1624340$. Thus, the client needs to send 1624340 bytes (≈ 1.5 mega bytes) for the encryption costs to equate the *ECDHE-ECDSA* handshake cots.

5 Conclusion

In our work, we decomposed TLS into individual parts and evaluated the cost of each security service. We have evaluated the cost of every TLS configuration available in *mbedTLS 2.7.0* and the underlying algorithms. The presented results can be used to make informed decisions about the security/cost trade-offs, specific to the environment.

The results presented here were obtained on a powerful, modern-day computer. Despite that, they are still relevant when considering the costs on constrained IoT devices. While on a different device, the absolute cost numbers will be different, they would still maintain a similar proportion one to another and follow a similar trend. Moreover, the developed tooling can be used to obtain profiling results on any machine, thus giving device-specific cost information. The formula used to obtain the CPU cycle count estimate can also be changed to one's needs.

5.1 Future Work

There are several metrics that would be interesting to analyze in future work. While we analyzed the cost in terms of CPU cycle estimates, would be interesting to analyze the actual number of CPU cycles used, by querying the processor's counters directly. Costs in terms of power usage and time taken are also important ones to explore. Finally, all of the mentioned metrics should be obtained on IoT hardware.

References

1. Addition to tls 1.3 contributors list. `https://github.com/tlswg/tls13-spec/commit/43461876882a60251ecf24fb097f0ce2d7be4745`. (Accessed on 01/11/2018).
2. E. Rescorla. The transport layer security (tls) protocol version 1.3. RFC 8446, RFC Editor, August 2018.
3. Nvd - cve-2018-1000520. `https://nvd.nist.gov/vuln/detail/CVE-2018-1000520`. (Accessed on 10/10/2018).
4. ssl_server.c and ssl_client1.c are using an sha-1 signed certificate. `https://github.com/ARMmbed/mbedtls/issues/1519`. (Accessed on 10/15/2018).
5. update test rsa certificate to use sha-256 instead of sha-1 by iluxonchik. `https://github.com/ARMmbed/mbedtls/pull/1520`. (Accessed on 10/15/2018).

6. Shahid Raza, Ludwig Seitz, Denis Sitenkov, and Göran Selander. S3k: Scalable security with symmetric keysdtls key establishment for the internet of things. *IEEE Transactions on Automation Science and Engineering*, 13(3):1270–1280, 2016.
7. Aamer Nadeem and M Younus Javed. A performance comparison of data encryption algorithms. In *Information and communication technologies, 2005. ICICT 2005. First international conference on*, pages 84–89. IEEE, 2005.
8. Sheena Mathew and K Poulose Jacob. Performance evaluation of popular hash functions. *World Academy of Science, Engineering and Technology*, 61:449–452, 2010.
9. Levent Ertaul, Anup Mudan, and Nausheen Sarfaraz. Performance comparison of aes-ccm and aes-gcm authenticated encryption modes. In *Proceedings of the International Conference on Security and Management (SAM)*, page 331. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2016.