Open-Source OPC UA Security and Scalability

Nikolas Mühlbauer*, Erkin Kirdan†*, Marc-Oliver Pahl‡*, Georg Carle*
*Technical University of Munich, †Covalion, Framatome, ‡IMT Atlantique
*{n.muehlbauer,kirdan,pahl,carle}@tum.de, † erkin.kirdan@framatome.com, ‡ marc-oliver.pahl@imt-atlantique.fr

***Abstract*—OPC UA is widely adopted for remote-control in industrial environments. It has a central role for industrial control systems as it enables remote management. Compromising OPC UA can lead to compromising entire production facilities. Consequently, OPC UA requires a high level of security. Major commercial OPC UA implementations have compliance certificates ensuring that their security models obey the specification. However, open-source OPC UA implementations that have wide deployment mostly lack these certificates. In this work, we investigate the security models of the four most commonly used open-source implementations: open62541, node-opcua, UA-.NETStandard, and python-opcua. Furthermore, their scalabilities for the number of clients and OPC UA nodes are also analyzed.**

***Index Terms*—OPC UA, security, scalability, open-source**

I. INTRODUCTION

Essential parts of the Industry 4.0 paradigm are, connecting previously unconnected systems, and allowing remote control. Since the 1980s, communication protocols that meet the complexity of industrial processes have been developed. Today, Open Platform Communications (OPC) Unified Architecture (UA) is the dominant protocol for monitoring and controlling industrial processes.

OPC UA is a platform-independent service-oriented architecture. It is maintained by the OPC Foundation [1]. OPC UA has use-cases across different hardware platforms such as personal computers, cloud-based servers, micro-controllers, or programmable logic controllers (PLC). OPC UA implementations run on diverse operating systems including Windows, Linux, and mobile operating systems.

OPC UA is a client-server architecture. The core of the protocol-specification is the *Address Space Model* [2] and the *Services* [3]. *Address Spaces* define a standard way to represent objects between servers and clients. OPC UA Managed Objects have variables and methods. An Address Space implements them as Nodes of different Node Classes.

Services are the standard-operations offered to clients by servers. Some examples are the *Discovery Service Set*, which allows a client to discover the Endpoints implemented by a server or the *SecureChannel Service Set* that enables clients to establish a communication channel with a server.

OPC UA can be used to connect industrial controllers in factories to the Internet. This process requires attention on all management levels and must be carried out in the best possible way. For example, without proper transition, previously high-security facilities that have rigorous physical access control could suddenly become exposed to the world. The example shows that the security aspect of OPC UA becomes essential. Possible damage caused by industrial devices in critical facilities, such as power grids, can be severe.

This research was funded by the German Federal Ministry of Economic Affairs and Energy (BMWI) in DECENT (0350024) and the German-French Academy in SCHEIF.

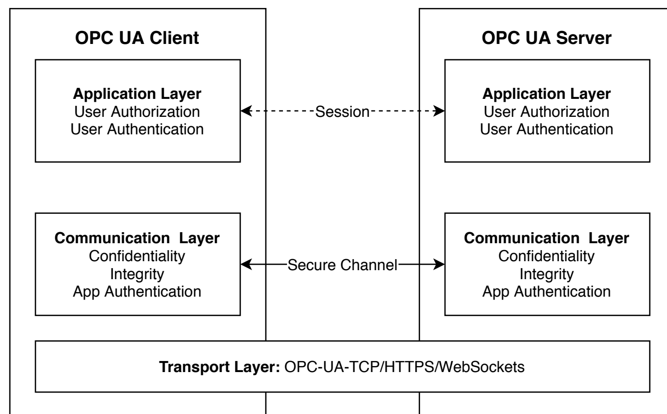

Fig. 1. OPC UA Security Model (based on [4])

The Security Model of OPC UA is designed carefully and plays an essential role in its popularity [4]. Figure 1 shows the OPC UA Security Model [4]. Sessions maintain *user authentication* and *user authorization* in the OPC UA *application layer*. A server allows a client to access its resources after the verification of a *user identity token* passed by client. *Sessions* are optional. A client can access some services stateless without establishing a session. *Secure Channels* ensure *confidentiality*, *integrity*, and *application authentication* in the OPC UA *communication layer*. The protocol requires to establish a secure channel before any interaction between a client and a server.

There are several implementations of OPC UA, both open-source and commercial. Different studies cover the security aspects of the OPC UA specification and different ways to elevate the security level. However, there is a research gap in the analysis of the open-source implementations. Due to their increasing popularity, insufficiencies in the security of open-source implementations can lead to real-life damage.

Therefore, this paper provides a security and scalability analysis of the most important open-source OPC UA implementations: open62541, node-opcua, UA-.NETStandard, and python-opcua.

Section II gives an overview of state of the art. Section III motivates the selection of the assessed open-source imple-

mentations. Section IV is the main contribution, containing a detailed code-review-based security analysis. Section V provides a scalability analysis, giving insights on how well each solution fits for larger deployments.

II. Related Work

In contrast to our work, the related work commonly focuses either on the security of the specification or outdated implementations. For performance measurements, the related work only targets one specific implementation rather than aiming at a comparison or focuses on different aspects than us.

The authors of [5] measure the server performance of subscriptions with varying sampling intervals and number of clients. They develop their servers and clients with Prosys OPC UA Java SDK. Data values are created on a different machine communicating with the OPC UA server via a network.

In [6] an OPC UA server for embedded devices is implemented in C targeting the OPC Foundation's "Nano Embedded Device Server profile". The authors achieve a binary size of 15 KB, including TCP/IP stack and minimal RAM usage when communicating with a client. No further performance details, such as CPU usage are given.

The authors of [7] promote the development of open-source OPC UA implementations and give a short comparison of all open-source implementations known at that time, including node-opcua and open62541, that are also investigated in our work. Furthermore, a detailed view of the development-state and potential use cases of open62541 are given. However, the projects evolved in the meanwhile, and no security analysis is provided.

In [8], a formal verification of the OPC UA protocol, as published in 2012 and 2013, is performed using ProVerif. The OpenSecureChannel sub-protocol had security issues such as not specifying that nonces and passwords must not be sent in plaintext when using security mode "Sign". Solutions for these issues are provided, and the FreeOpcUa (C++) implementation is analyzed, which is not vulnerable, according to the authors.

An overview of all known OPC UA implementations is given in [9], including commercial and open-source implementations. The features such as licence or supported transport protocols, security policies or services are compared. However, these results are partially outdated by now since some of the projects advanced like open62541. On the other hand, some projects like FreeOpcUa (C++) have been stopped.

In [10], the OPC UA specification, version 1.02 is analyzed, and the authors find no systematic flaws. Additionally, they investigate the security of the former reference implementation of the OPC UA Foundation written in ANSI C. This implementation is succeeded by UA-.NETStandard that is included in our analysis.

In [11], the performance of the OPC UA publish-subscribe architecture is investigated. Measurements on a Raspberry Pi Zero identified that CPU power is most crucial for performance. In contrast, our work does not focus on publish-subscribe communication since there are not as many implementations as for server-client communication.

The authors of [12] compare the performance of three OPC UA implementations, including open62541 and python-opcua, that are also covered in our work. As the focus lays on embedded systems, the measurements such as for RTT are done on two Raspberry Pi 3 B+. The authors inspect the memory requirements for variable nodes, like in our paper, but not the CPU consumption. In contrast to us, the scalability with increasing number of clients is not investigated and also, there is no security analysis done.

III. Open-Source Implementation Selection

There are both proprietary and open-source implementations of OPC UA. They differ in the number of users, languages, and supported features. We consider the following selection criteria in our research.

1) **Completely open-source:** some implementations provide a wrapper over proprietary software. To be independent of commercial software, we only consider completely open-source implementations.
2) **Active:** We only consider the implementations that are still actively improved, or at least maintained, because deprecated implementations will become vulnerable, incompatible and loose popularity.
3) **Full-stack server implementation:** some implementations are only simulators or client implementations. We only consider full-stack OPC UA server implementations, as these cover most use-cases.

Several implementations fit our criteria. We have selected the four most commonly used implementations among them; open62541 [13], node-opcua [14], UA-.NETStandard [15] and python-opcua [16]. We determined their popularity based on the star counters in their GitHub pages and verified the ranking using trends in web search engines. The 5th most-popular implementation is far behind with 482 stars. Table I shows the implementations with their languages and the versions used for the analysis and measurements. Furthermore, we also give their GitHub stars as of 2020/07/14.

TABLE I
Most common open-source OPC UA implementations

Implementation	Language	Version	Stars
open62541	C	1.0.1	1205
UA-.NETStandard	C#	1.4.359.31	867
node-opcua	JavaScript	2.4.4	866
python-opcua	Python	0.98.9	717
(Eclipse Milo)	(Java)	(0.3.8)	(482)

IV. Security Analysis

The second part of the OPC UA specification describes the security model [4]. Clause 6 provides implementation and deployment considerations for developers and users. It is the starting point of our analysis. We focus on the aspects related to the implementation. Furthermore, we add two necessary aspects, 1) supported security policies, and 2) dependencies. Together, this results in the following overall security aspects that we analyze in the implementations.

1) **Dependencies:** A security issue in a dependency becomes a security issue of the implementation. Therefore, we inspect the security-critical dependencies, i. e., crypto libraries and some parsers.
2) **Timeouts:** Carefully-set timeouts reduce the risk of Distributed Denial of Service (DDoS) attacks and lower resource requirements. Important timeouts in OPC UA are the Session and Channel timeouts. They are the maximum time that a Session and a Secure Channel can be idle. Another critical timeout is the lifetime of the SecurityToken.
3) **Security Policies:** The list of available Security Policies is relevant because some are deprecated. The specification defines the following policies [17]:
   - None,
   - Basic128Rsa15 (deprecated),
   - Basic256 (deprecated),
   - Basic256Sha256,
   - Aes128_Sha256_RsaOaep, and
   - Aes256_Sha256_RsaPss.

   Each Security Policy is used in one of the Security Modes [4]:
   - None,
   - Sign, or
   - SignAndEncrypt.

   The security mode "None" is only applicable if security policy "None" is used. Vice versa, none of the other security policies can use the security mode None.
4) **Message Processing:** The structure of the received messages should be checked before processing. Malformed packets should be discarded or result in error messages. The specification defines the following message types [18], split into Connection Protocol Messages:

   HEL sent by a client after TCP handshake to open a connection,
   ACK sent by the server to confirm HEL messages,
   ERR sent in case of a fatal error, and
   RHE used for reverse connect, i. e., server initiates a connection

   and Secure Conversation Messages:

   OPN sent by a client to open a SecureChannel,
   MSG used for messages sent over SecureChannel, and
   CLO sent to close a SecureChannel.

   Secure messages of MSG type can be split into chunks, indicated by the isFinal value. It can be either F (final chunk), C (intermediate chunk), or A (abort) in case of an errors.
5) **Randomness:** The quality of random number generators has a high impact on security.
6) **Miscellaneous:** Findings that do not fit any other aspect but are still of security interest are summarized here.

Moreover, there are some known issues in the assessed source code with comments on how to address them since all implementations are actively developed. We mention the important ones despite that they are likely to be solved in the near future.

A. *open62541*

open62541 is written in C (C99 standard) language. Thus, it can be used on multiple platforms. However, some plugins are only provided for specific operating systems. The authors of open62541 focus on the quality of the code and therefore have minimum requirements for a release [13], such as no errors with OPC UA Compliance Testing Tool (CTT) [19], no compiler warnings and no indications from various analysis tools.

1) Dependencies: The project uses the following external libraries or code snippets:

- PCG random number generator [20] and
- Mbed TLS [21]

The authors implement their own json parser.

2) Timeouts: Table II shows the default and maximum values of the timeouts. All timeouts can be set in millisecond resolution. The No-HEL timeout specifies how long the server waits for a client to send a HEL message after the TCP handshake. If a server does not respond within the client timeout, the client's connection fails.

TABLE II
TIMEOUTS OF OPEN62541

Timeout	default	max
Session	1 h	1 h
SecureChannel	10 min	-
SecurityToken	10 min	10 min
No-HEL	2 min	1 h
Client Timeout	5 s	-

3) Security Policies: The following security policies are supported:

- Basic128Rsa15,
- Basic256,
- Basic256Sha256 and
- None.

The support for encryption is disabled by default but can be enabled by setting a preprocessor constant, i. e., user needs to recompile the application to use security mode SignAndEncrypt.

4) Message Processing: Only complete messages are processed. Messages are not processed after a channel gets closed. All message chunks with invalid types lead to the sending of an ERR message and the closing of the channel. The session timeout is checked every time a MSG is received.

5) Randomness: The `mbedtls_ctr_drbg_random` function of Mbed TLS is used for nonces. The session nonce length is 32 Byte. The PCG random number generator is used for less security-critical random values, such as Guids and EventIds.

6) Miscellaneous: Data structures of secure channels are initialized with all zeros. A discovery mechanism with optional multicast support is implemented. The error codes for revoked certificates are hidden from clients. According to the code comments, the developers plan to hide some other error codes from clients as well.

B. node-opcua

This implementation is written in javascript and typescript and can be executed in the Node.js runtime [22].

1) Dependencies: The project relies on more than 1000 external node modules [23]. The crypto module from Node.js [24] is used for cryptographic operations.

2) Timeouts: Table III shows the default, minimum and maximum values of the timeouts. They can be specified in millisecond resolution. The TCP HEL/ACK timeout specifies the time a server waits for an ACK message. When the initial connection to a Local Discovery Server (LDS) fails, it tries infinitely with doubling the delay from 2 to the maximum of 50 seconds between the attempts. Then the server registers itself again every 8 minutes to the LDS.

TABLE III
TIMEOUTS OF NODE-OPCUA

Timeout	default	min	max
Session	30 s	100 ms	50 min
SecurityToken	10 min	-	-
TCP HEL/ACK	30 s	-	-
RegistrationServer	8 min	-	-

3) Security Policies: The following security policies are supported:

- Basic128Rsa15,
- Basic256,
- Basic256Sha256, and
- None.

Security policy and mode "None" are assumed when they are undefined. The public key length of certificates must be one of 1024, 2048, 3072 or 4096.

In OPC UA, before a client can use a session, a CreateSession request that contains nonces for the security policies [3] must be sent. Then, the client must request the activation of the session. These ActivateSession requests can also be used to change the underlying SecureChannel of a session.

In the node-opcua server, this procedure is implemented as follows: When a client requests the creation of a session, the client nonces are checked and stored such that the same client cannot reuse them. Furthermore, the Uniform Resource Identifier (URI) of the application provided by the client to create a session must match with the URI in the client certificate. Endpoint URIs are not checked whether they belong to the server hostnames or not. It is a known issue the developers plan to fix.

The activation of Sessions and the renewal of SecureChannels follow the specification: The same SecureChannel used for the creation of a Session must be used when a client requests the activation of the Session. Sessions get corrupted when they are used before they have been activated. Additionally, when renewing a a SecureChannel, the client must use the same certificate for the old and new SecureChannel and the same identity token stored in the Session.

4) Message Processing: The function `_read_headers` in `message_builder.ts` checks, whether the header has the correct size (12 Byte)before processing. The HEL and ACK messages lead to the setting of SecurityPolicy "None". ERR messages are ignored during parsing. If the debug flag is set, the respective error code and the message are printed. When an OPN message is received, the SecurityPolicy is set according to its asymmetric security header.

In node-opcua, there is no check whether the message type is valid. Instead, the decryption function `message_builder._decrypt` assumes the type MSG. The function `_feed_message_chunk` gets called when a chunk is received. It reads the header and checks for the final field if it is F (Final), C (Chunk) or A (Abort). An error is returned if it is "Abort". The channel identifier (ID) in a message is checked against the expected channel ID.

5) Randomness: As stated in the specification, server and client nonces have the same length as the symmetric key. They are generated using the `randomBytes` function of the crypto module. When a client request a subscription, the server draws the initial subscription ID uniformly random from (0,1000000) using `Math.random` and increments it by one for each following ID.

6) Miscellaneous: During the installation of node-opcua, 1398 vulnerabilities contained in the dependencies are shown. However, almost all of them can be fixed by updating the dependencies.

When a certificate or a private key are not specified, a 2048 bit self-signed certificate is used that has been created together with other certificates at installation time.

Unused channels are closed. A delay of 2 milliseconds for the processing of FindServersRequest is applied to mitigate DDoS attacks. DNS-Based Service Discovery [25] is available as an alternative to LDS. The auditing functionality overwrites only 13 digits of the passwords with "*" which could result in partial leak of passwords. Anonymous connections are permitted by default. Server methods can be either hidden by default, i. e., they can not be discovered by clients or exposed by mDNS or via an LDS. The asymmetric signature length is taken from the certificate length. Checks whether Certificates are revoked or contain the same URI as in the *ApplicationDescription* are not implemented, but this is planed to be solved.

Due to missing support for adding/deleting Nodes and References, clients cannot create a node in the server remotely.

C. python-opcua

There is a fork called opcua-asyncio [26] that is supposed to be the successor of python-opcua. However, we decided to investigate python-opcua since opcua-asyncio is at early stage of implementation.

1) Dependencies: The project uses the following modules [27]:

- sqlite3,
- urllib.parse,
- xml and
- base64.

These modules are all part of the python standard library. For cryptographic operations, the cryptography package [28] is used.

2) Timeouts: Table IV shows the default and the minimum values of the timeouts. Note that the server uses the same timeout for sessions and SecureChannels set in ms granularity. The server registers itself every minute to the LDS by default. This timeout can be set in second resolution.

TABLE IV
TIMEOUTS OF PYTHON-OPCUA

Timeout	default	min
Server (Session and SecureChannel)	1 h	10 s
Registration server	1 min	10 s

3) Security Policies: The following security policies are supported:

- Basic128Rsa15 (disabled by default),
- Basic256 (disabled by default),
- Basic256Sha256, and
- None.

Anonymous, certificate and username options are available for the client identification, but no tokens.

4) Message Processing: When parsing an ACK, HEL or ERR message, the error messages are logged as a warning. After receiving an OPN message, the security policy is set according to the asymmetric security header. After receiving MSG and CLO messages, symmetric security headers are checked. Close messages lead to the closing of a connection. Every other message type results in an error. Before a chunk is appended to a message, its sequence number is checked whether it conforms to the specification.

5) Randomness: Nonces have 32 Bytes. They are created using `os.urandom`.

6) Miscellaneous: A python shelve [27] that caches values on the file system, is offered to make the start-up faster on embedded devices. History uses either a in-memory python dictionary or an SQLite database [29]. The URI used during the registration must belong to the known servers. This mitigates DDoS attacks.

There are predefined users: Admin, Anonymous and User. The remote admin is enabled by default with an empty password enabling a client to add or delete nodes. Password encryption can be ”None”, ”rsa-1_5”, or ”rsa-oaep”.

SetMonitoringMode and SetPublishingMode requests are currently not implemented. Unsupported requests result in an error.

D. UA-.NETStandard

The OPC UA Foundation develops the UA-.NETStandard in C# for the .NET Framework [30] as the reference implementation. It is the only implementation that supports both TCP and HTTPS transport protocols and binary and XML encodings.

1) Dependencies: The project uses the following namespaces that are part of the .NET API [31]:

- System.Security.Cryptography,
- System.xml and
- Newtonsoft.Json.

2) Timeouts: Table V shows the default, minimum and maximum values of the timeouts. All timeouts are specified in milliseconds and can be configured using XML files or `ApplicationConfiguration.cs`. When using subscriptions, the lifetime is limited. Requests with timestamps older than maxRequestAge cause a timeout error. In the beginning, a server tries to register to a discovery server. If this fails, the server repeats it with a doubling interval from 1 second to the maximum of 30 seconds.

TABLE V
TIMEOUTS OF UA .NET STANDARD

Timeout	default	min	max
Session	1 min	10 s	1 h
SecureChannel	10 min	-	-
SecurityToken	1 h	1 min	-
SubscriptionLifetime	-	10 s	1 h
MaxRequestAge	10 min	-	-
Registration server	1 s	1 s	30 s

3) Security Policies: The following security policies are supported:

- Basic128Rsa15,
- Basic256,
- Basic256Sha256 (default),
- Aes128_Sha256_RsaOaep,
- Aes256_Sha256_RsaPss, and
- None.

Basic256Sha256 is used in SignAndEncrypt mode by default.

UA.Net aggregates SecurityPolicys and SecurtiyModes into a so-called ServerSecurityPolicy as these two are always used together. Typically, different ServerSecurityPolicies can be used with different clients in different connections. The server owner provides the list of supported policies to launch a server. However, if HTTPS is used as transport protocol, only one of the ServerSecurityPolicies in the list is selected: the server selects the first policy with SignAndEncrypt mode, or the first policy if none of them features SignAndEncrypt. This behaviour might lead to the selection of a sub-optimal SecurityPolicy. Additionally, the implementation features multiple levels of strictness for certificate trust. When using HTTPS as the transport protocol, the revocation checks for TLS certificates are disabled, and no TLS client certificates are used.

4) Message Processing: The type of message is checked after every reception. Non-valid types cause an ERR message and the closing of the TCP connection. Messages and chunks can be checked for validity. HEL, ACK or ERR messages are assumed valid. The chunk type field is also checked for MSG, OPN, and CLO messages. The message is assumed valid only if the chunk type field is F (Final) or I (Intermediate). There is also a separate function that checks if a message is of type A (Abort).

5) Random Numbers: Random numbers for nonces and certificates are generated using the `RandomNumberGenerator` of `System`. The length of nonces is 32 Bytes.

6) Miscellaneous: Application certificates are generated during the first startup. It is more secure than not using certificates at all, but it introduces the risk of low entropy on embedded devices [4].

The server supports OAuth 2.0 [32] and Multicast DNS [25], but the latter is disabled by default.

E. Discussion

There are differences in the number of dependencies among the implementations, partly caused by their different languages. While open62541 only relies on an external crypto library and implements the parsers, the other implementations use more dependencies. As long as the dependencies are kept up-to-date and patched, these projects are not less secure since they all use tested and proven libraries. However, as in real-life patching is often neglected, node-opcua has a higher risk due to the many dependencies.

For the timeouts, all implementations provide suitable default values. Due to different possible use cases, the default Session timeout ranges between 30s and 1h. Unfortunatly, python-opcua does not provide the maximum values for timeouts. This could result in DoS attacks, when clients constantly create sessions but never use them. The UA-.NETStandard provides the maximum number of configurable timeouts and also the most convenient way to set these values.

As the reference implementation of the specification, UA-.NETStandard offers all security policies targeting client-server communication. The rest of the projects allow for at least one policy that is considered secure. A convenient automatic creation and usage of certificates is available in node-opcua and UA-.NETStandard, which helps unexperienced users to set up a secure server. We recommend using these two implementations to get familiar with OPC UA. In contrast, open62541 disables encryption by default, and requires a re-compilation to enable it. This might be a pitfall for users. Node-opcua accepts revoked certificates in the version we are investigating but a recent github commit indicates, that it will be fixed in upcoming releases.

Message processing is done similar among implementations. Only node-opcua misses the check for invalid message types and assumes them valid, which is not recommended. None of the implementations supports RHE messages. Therefore reverse-connect cannot be used.

Random numbers for nonces are generated either by using functions of the crypto library or the operating system in the case of python and .NET, which we both consider secure.

Overall, we consider UA-.NETStandard and open62541 the most secure implementations and indicate smaller issues in node-opcua and python-opcua. One may stress, that the improper use or configuration of any of the investigated projects can also cause security issues. UA-.NETStandard and node-opcua provide the most secure and strict default settings.

V. Scalability Evaluation

For the deployment of an OPC UA server, one should consider how good it can cope with the expected load and how many resources it consumes. Therefore, we perform the following tests to investigate the scalability of the implementations.

1) The number of clients: OPC UA servers often have to deal with many clients in parallel. For this reason, we investigate how well the implementations scale with an increasing number of clients. A reasonable assumption is that clients open a session and regularly request data, using the same session. This forces the server to keep state information for each of the clients. Thus, we measure the resource usage on the server with an increasing number of clients. The client applications open a connection to the server, and periodically accesses the server time.
2) The number of nodes: Another scenario is that a server needs to store many nodes. We measure resource usage on the server with an increasing number of nodes. The nodes added to the server contain only one numeric variable.

All tests are performed on a Linux computer equipped with an Intel(R) Core(TM) i3-4030U CPU at a frequency of 1.90GHz. Connections between clients and servers are made using the loopback-interface. All the measurements are taken using the GNU time tool [33].

A. Number of clients

We implement OPC UA clients that connect to the OPC UA server of the same implementation and then read the value of the server-time every second for a minute. We keep the programs as simple as possible and set the security policy "None" to find out the minimum resource requirements. We slightly adapt `tutorial_server_firststeps.c` and `client_connect_loop.c` of open62541 and implement similar servers and clients for the other implementations.

The clients start, once the server is ready. After 60 server-time requests, clients close the connection, and the server gets closed. Since the servers need some seconds until they are ready, the execution time between server and client slightly differs in each implementation.

Figure 2 shows the resource usages of the servers. Tests were performed for 0, 10, 20, 30 and 40 clients. CPU time is given in seconds and memory usage in MB. Test-runs only differ negligibly. Therefore, only the results of the last run are used. open62541 uses the least amount of resource since it

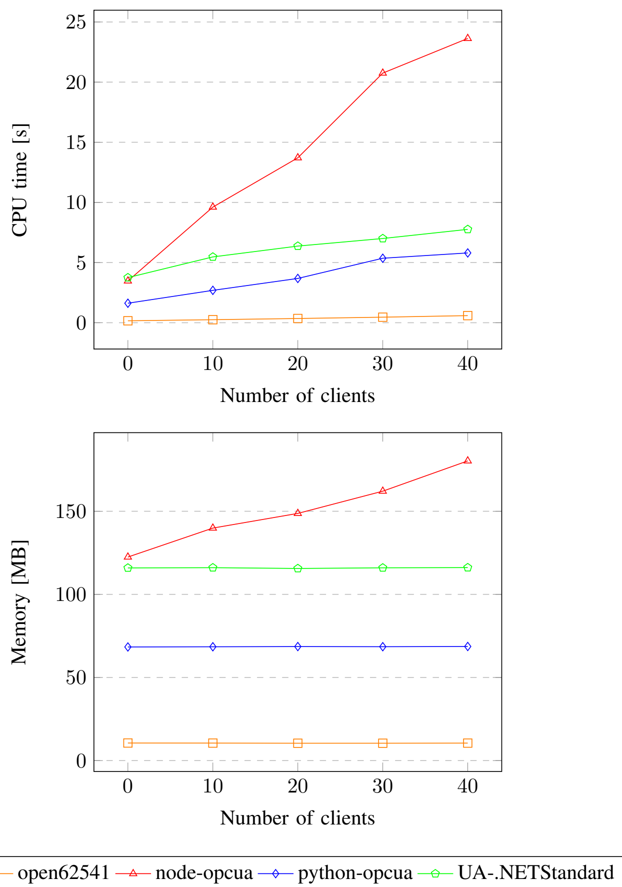

Fig. 2. Resource usage with increasing number of clients

does not rely on any interpreters or runtime environments. Its CPU time is 0.16s with no client and grows slightly above linear by about 100ms per 10 clients, while the memory usage is constant at around 10.5MB. python-opcua and UA-.NETStandard have the second and the third least usages. They follow a similar pattern with a constant memory usage of 68.5MB and 116MB respectively as well as a linear increase of about 1s per 10 clients of CPU time, starting at 1.6s and 3.8s respectively. node-opcua has similar base requirements (with no client) as UA-.NETStandard. However, it scales much worse. It needs about 5s more CPU time and 15MB additional memory per 10 clients. This makes it the only implementation with increasing memory requirements.

We also give the resource usages of the clients in Table VI. The client results have a similar pattern to the servers of the same implementations.

TABLE VI
MEMORY AND CPU USAGE OF CLIENTS

Implementation	Memory	CPU time	CPU load
open62541	3MB	0.02s	0.034%
node-opcua	94MB	1.89s	3.05%
python-opcua	37MB	0.39s	0.714%
UA .NET	67MB	1.47s	2.37%

B. Number of nodes

We implement servers that create nodes with a single 32-bit integer variable to find the node limitations of the servers. Our aim is the CPU time until the server is ready for clients, and the peak memory usage for storing the nodes. Therefore, we kill the servers shortly after they created the nodes. Since the servers add the values as fast as possible, the CPU load is close to 100% during start-up. Thus, we are interested in CPU time rather than the CPU load.

Figure 3 shows the resource usages. Tests were performed for 0 nodes up to 50k nodes, in 10k steps. CPU time is given in minutes and memory usage in MB. Since all test runs only differ negligibly, the results of the last run are used. At first look of the results, open62541 and UA-.NETStandard scale much better than python-opcua and node-opcua.

Again, open62541 has the lowest resource usages. It has a base requirement of 0.11s CPU time and 10.5MB memory. They increase per 10k nodes by about 200ms and 10MB respectively. UA-.NETStandard has the second-lowest usages with the base CPU time of 2.6s and the base memory of 119MB. However, it scales even better than open62541 with only about 100ms increase per 10k nodes and constant memory usage for up to 20k nodes, which increases then up to 137MB for 50k nodes, i. e. on average 6MB per 10k nodes.

python-opcua and node-opcua show an exponential growth of CPU time. While python-opcua has a lower CPU time usage of 1.6s compared to 4.7s of node-opcua in the beginning, it scales worse. For the first 10k nodes, python-opcua needs 88s, and node-opcua needs 64s. For 20k nodes, python-opcua requires 368s as opposed to 250s of node-opcua. At the maximum of 50k nodes, the difference between them is more than 10 min. For memory usage, they both show a linear increase. python-opcua starts at 69MB and shows a constant increase of about 145MB per 10k nodes. node-opcua starts at a higher base memory usage of 150MB but shows in average a smaller increase of 125MB per 10k nodes.

C. Discussion

Overall, open62541 uses the least resources for the same operations. The reason is that it does not rely on any interpreters or runtime environments. It scales best for client connections, whereas UA-.NETStandard scales better for nodes. While python-opcua still scales good for increasing client connections, it scales worst for node generation. Similar to python-opcua, node-opcua also depends on runtimes and thus scales poor in both tests, especially when it comes to the generation of nodes. Overall, only open62541 and UA-.NETStandard are suitable for both high numbers of clients and nodes.

VI. CONCLUSION

This paper analyzed the security and scalability of the four currently most-important OPC UA open-source implementations (section III). The security analysis based on a detailed code-review. The results indicate that the projects offer at least one of the currently recommended *SecurityPolicy* and handle the critical Randomness appropriately (section IV). The

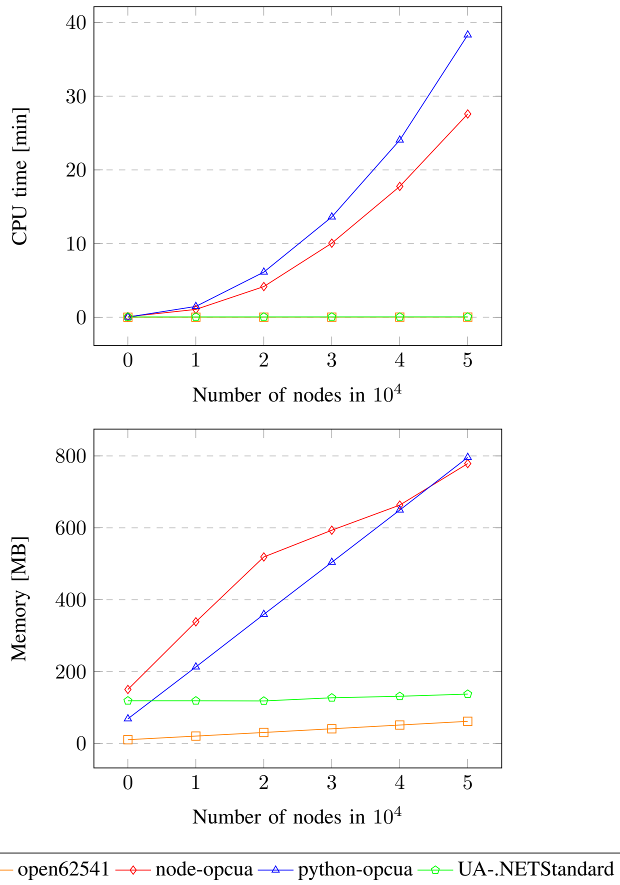

Fig. 3. Resource usage with increasing number of nodes

identified significant shortcomings are missing timeouts and the processing of messages. Our scalability tests (section V) show significant differences between implementations. The key factor here is the differences in the programming languages.

Overall, our survey shows that the open-source implementations considered provide very good security standards already. They can be therefore indeed an alternative to the commercial implementations also from the security point-of-view.

There are still shortcomings such as missing upper timeout limits in python-opcua and less strict packet type checks in node-opcua, that should be fixed. For open62541, UA.NETStandard, we did not identify relevant security flaws.

Considering our two criteria, security and scalability, open62541 and UA-.NETStandard achieve the best results.

References

[1] OPC Foundation, *Unified Architecture*. https://opcfoundation.org/about/opc-technologies/opc-ua Accessed 08 May. 2020.

[2] OPC Foundation, *OPC UA Specification Part 3: Address Space Model*, 1.04 ed., 2017.

[3] OPC Foundation, *OPC UA Specification Part 4: Services*, 1.04 ed., 2017.

[4] OPC Foundation, *OPC UA Specification Part 2: Security Model*, 1.04 ed., 2018.

[5] M. Fojcik and K. Folkert, “Introduction to opc ua performance,” *Computer Networks and Isdn Systems*, pp. 261–270, 2012.

[6] J. Imtiaz and J. Jasperneite, “Scalability of opc-ua down to the chip level enables “internet of things”,” in *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, pp. 500–505, 2013.

[7] F. Palm, S. Gruner, J. Pfrommer, M. Graube, and L. Urbas, “Open source as enabler for opc ua in industrial automation,” in *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, pp. 1–6, 2015.

[8] M. Puys, M.-L. Potet, and P. Lafourcade, “Formal analysis of security properties on the opc-ua scada protocol,” in *International Conference on Computer Safety, Reliability, and Security*, pp. 67–75, 2016.

[9] H. Haskamp, M. Meyer, R. Mollmann, F. Orth, and A. W. Colombo, “Benchmarking of existing opc ua implementations for industrie 4.0-compliant digitalization solutions,” in *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*, pp. 589–594, 2017.

[10] “Opc ua security analysis,” tech. rep., Federal Office for Information Security (BSI), 2017.

[11] A. Burger, H. Koziolek, J. Rückert, M. Platenius-Mohr, and G. Stomberg, “Bottleneck identification and performance modeling of opc ua communication models,” in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pp. 231–242, 2019.

[12] A. Cenedese, M. Frodella, F. Tramarin, and S. Vitturi, “Comparative assessment of different opc ua open–source stacks for embedded systems,” in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1127–1134, IEEE, 2019.

[13] J. Pfrommer and S. Profanter, *open62541*. https://github.com/open62541/open62541 Accessed 08 May. 2020.

[14] E. Rossignon, *node-opcua*. https://github.com/node-opcua/node-opcua Accessed 08 May. 2020.

[15] OPC Foundation, *UA-.NETStandard*. https://github.com/OPCFoundation/UA-.NETStandard Accessed 08 May. 2020.

[16] Free OPC-UA Library, *python-opcua*. https://github.com/FreeOpcUa/python-opcua Accessed 08 May. 2020.

[17] OPC Foundation, *OPC UA Specification Part 7: Profiles*, 1.04 ed., 2017.

[18] OPC Foundation, *OPC UA Specification Part 6: Mappings*, 1.04 ed., 2017.

[19] OPC Foundation, *OPC UA Compliance Test Tool (UACTT)*. https://opcfoundation.org/developer-tools/certification-test-tools/opc-ua-compliance-test-tool-uactt/ Accessed 08 May. 2020.

[20] M. E. O'Neill, “Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation,” Tech. Rep. HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, 2014. https://www.cs.hmc.edu/tr/hmc-cs-2014-0905.pdf.

[21] Trusted Firmware, *Mbed TLS*. https://tls.mbed.org/ Accessed 08 May. 2020.

[22] OpenJS Foundation, *Node.js JavaScript runtime*. https://nodejs.org/ Accessed 08 May. 2020.

[23] E. Rossignon, *node-opcua dependencies*. https://github.com/node-opcua/node-opcua/blob/master/package-lock.json Accessed 08 May. 2020.

[24] Node.js, *Node.js v14.0.0 Documentation*. https://nodejs.org/api/crypto.html Accessed 08 May. 2020.

[25] S. Cheshire and M. Krochmal, “Multicast dns,” RFC 6762, RFC Editor, February 2013. http://www.rfc-editor.org/rfc/rfc6762.txt.

[26] Free OPC-UA Library, *opcua-asyncio*. https://github.com/FreeOpcUa/opcua-asyncio Accessed 08 May. 2020.

[27] Python Software Foundation, *The Python Standard Library*. https://docs.python.org/3.9/library/ Accessed 08 May. 2020.

[28] Python Cryptographic Authority, *Cryptography package*. https://cryptography.io/ Accessed 08 May. 2020.

[29] The SQLite Consortium, *SQLite*. https://sqlite.org/ Accessed 08 May. 2020.

[30] Microsoft, *.NET Framework*. https://dotnet.microsoft.com/ Accessed 08 May. 2020.

[31] Microsoft, *.NET API*. https://docs.microsoft.com/en-us/dotnet/api/ Accessed 08 May. 2020.

[32] IETF OAuth Working Group, *OAuth*. https://oauth.net/2/ Accessed 08 May. 2020.

[33] Free Software Foundation, *GNU Time*. https://www.gnu.org/software/time/ Accessed 08 May. 2020.