Testing IoT Protocol Requirements Using Fuzzing and Symbolic Execution: Application to CoAP

Hooman Asadian*, Paul Fiterău-Broştean*, Bengt Jonsson*, and Konstantinos Sagonas*†
*Department of Information Technology, Uppsala University, Uppsala, Sweden
†School of Electrical and Computer Engineering, National Technical University of Athens, Athens, Greece

***Abstract*—The rapid expansion of the Internet of Things (IoT) has introduced the need for thorough testing of its protocol implementations to ensure conformance to their specifications and increase their security. This paper investigates the application of two of the major testing techniques, *fuzz testing* and *symbolic execution*, to test the implementations of the Constrained Application Protocol (CoAP), a key protocol in the IoT ecosystem. We explore the efficacy of these techniques in discovering requirement violations in CoAP implementations. Focusing on two widely-used CoAP implementations, libcoap and FreeCoAP, we systematically apply both fuzzing and symbolic execution to test conformance with key requirements derived from CoAP's specifications. Our findings demonstrate the strengths and limitations of both approaches, and highlight nine non-conformances in these implementations, most of which have been fixed. Finally, we provide insights into how fuzzing and symbolic execution can be effectively utilized for protocol testing.**

I. Introduction

Network protocols enable reliable communication between different software and hardware components in most infrastructures of our society. Ensuring that the implementations of these protocols conform to their specifications is critical. Non-conformance can lead to security breaches, such as the Heartbleed [8] and the TLS POODLE downgrade [16] vulnerabilities, both of which were caused by improper handling of protocol-specific requirements.

With the rapid expansion of the Internet of Things (IoT), similar attention must be given to quality assurance of its underlying infrastructure; this includes testing of protocol implementations developed specifically for IoT systems. Several communication protocols, including MQTT, CoAP, and EDHOC, have been designed specifically for IoT environments, such as low-power devices operating over lossy networks. Implementations of such protocols merit particular testing effort, since sometimes they are deployed on devices that are not easy to access or update upon the discovery of errors. In this paper, we focus on testing implementations of the CoAP protocol [23], which plays a crucial role in the IoT ecosystem. CoAP follows a client/server interaction model similar to that of HTTP, but it is specifically optimized for machine-to-machine interactions.

Techniques for testing software and communication protocols have advanced dramatically in the last two decades. One major class of software testing focuses on detecting runtime errors, such as crashes and memory access errors, using techniques such as *fuzz testing* (*fuzzing*) and *symbolic execution* (*SE*). These testing techniques have proven very effective at identifying critical bugs, e.g., in systems libraries and device drivers, but have not been so much used to guarantee that protocol implementations conform to their specifications. That is the purpose of *conformance testing*, which examines whether a protocol implementation faithfully follows defined standards; this is essential to prevent subtle but potentially dangerous deviations that could compromise reliability and security. Traditional conformance testing techniques are less effective in uncovering bugs than fuzzing and symbolic execution; for this purpose some researchers are developing adaptations of fuzz testing [15] and symbolic execution [22], [24], [2], [3] that directly check whether a protocol implementation satisfies (parts of) its specification, e.g., as formulated in its RFC.

This paper investigates the application of fuzzing and symbolic execution to the challenge of testing CoAP implementations against its main RFCs. Both techniques aim to check how a system under test (SUT) responds to a wide range of different inputs. Fuzzing achieves this by generating a large number of (random) test inputs, constructed by mutating previously applied inputs. Using cleverly designed mutation strategies, it can often —but not always— quickly penetrate into the many corners of a SUT's code. Symbolic execution achieves a similar goal by systematically exploring a range of code paths followed by the program. It designates part of the test input as *symbolic*, and explores the code paths that are possible for different values of these symbolic inputs. Technically, this relies on instrumenting the SUT's code to enable symbolic manipulation of executed program statements. Against the above background, this paper:

- investigates the effectiveness of (adaptations of) fuzzing and symbolic execution in discovering bugs and requirement violations in IoT protocol implementations;
- compares fuzzing and SE in terms of their effectiveness at exposing bugs, as well as in time that this requires;
- (as a by-product) improves the quality of two widely used CoAP implementations by exposing bugs in them which have been fixed and/or reported to their developers; and
- offers some advice on how to apply these two techniques to IoT protocol implementations.

In our investigation, we have tested two well-known CoAP implementations written in C: libcoap [12] and FreeCoAP [9]. To assess their conformance, we employ twelve core requirements from the RFC documents that define the CoAP standard. In order to produce an unbiased comparison of fuzzing and SE, we have created a common test harness for both these techniques,

which has a common mechanism for checking a requirement, and which can be re-targeted to either technique by minimal modifications. The results indicate that both fuzzing and SE are very effective in locating protocol requirement violations fast; they have managed to uncover a total of nine bugs in these two implementations, the majority of which have been fixed by now and the rest have been reported to their developers.

The rest of this paper is organized as follows: After some background on fuzzing and SE, §III overviews CoAP, §IV details our methodology for applying fuzzing and SE to CoAP implementations, and §V presents our evaluation and findings. Finally, §VI concludes with a brief discussion.

II. Fuzzing and Symbolic Execution

In this section, we overview fuzzing and symbolic execution and recent works applying them to protocol implementations.

Fuzz Testing (Fuzzing) has established itself as a powerful technique for uncovering software bugs by mutating and executing numerous inputs to trigger and observe unintended behaviors, particularly in the context of memory errors and crashes. In the last decade, this area has been led by coverage-guided greybox fuzzing tools such as American Fuzzy Lop (AFL) [27] and its successor AFL++ [10], which offer advanced mutation strategies and enhanced performance, and have been used across a wide range of applications. Fuzzing tools also have been applied to test IoT protocol implementations of IPv6 network stacks (e.g., [20]). Other studies [13], [14], [28] are specifically aimed at fuzz testing of CoAP implementations. Several recent works propose techniques that tailor greybox fuzzing to stateful systems such as communication protocols [19], [17], [4]. A common characteristic of all these works is their primary emphasis on uncovering runtime errors (e.g., crashes) and robustness issues, rather than detecting non-conformance to the protocol's specification.

Symbolic Execution (SE), first introduced in the 1970s [11], has evolved into a powerful technique for exhaustive software testing [7], enabling systematic exploration of program paths by representing (selected) inputs symbolically rather than as concrete values. As a result, SE can uncover bugs, security vulnerabilities, and potential failures that might be missed by traditional testing methods. Applications of symbolic execution for testing network protocol implementations have been limited. Several works [22], [24], [18], [21], [2], [25], [3] target network protocol implementations by enabling test setups to designate specific fields in incoming packets as symbolic, allowing for the exploration of code paths that are reachable with different values for these fields. To enhance test coverage, some works [26], [25] incorporate the protocol's state machine into symbolic execution to account for the protocol's statefulness. However, the application of these approaches is limited to detecting runtime errors, and they are not utilized for checking protocol requirements. KleeNet [22] enables the checking of correctness properties by manually adding assertions to the SUT's source code. Symbolic variables are employed to simulate occurrences of packet loss and node failures, prompting SE to explore various error scenarios. The inserted assertions validate the consistency of the distributed state. SymbexNet [24] tests protocol implementations by first employing SE to generate test inputs that explore a wide range of code paths. These inputs are then replayed on the SUT, observing potential violations of rules derived from the protocol specification. Symbolic execution might miss requirement-violating input, when this input exercises the same code path on the SUT as non-violating input. Recently, we have proposed a technique that overcomes this issue by manually embedding assertions and assumptions into the SUT's source code [2]. The net effect is to guide the SE engine to investigate specifically code paths exercised by packet sequences that can violate protocol requirements. Follow-up work [3] concentrates the requirement testing logic into monitors, which are implemented in a component that is external to the SUT. Monitors provide a uniform mechanism for checking protocol requirements and alleviate the need for extensive SUT modifications.

III. Constrained Application Protocol

The Constrained Application Protocol (CoAP), as standardized in RFC 7252 [23], is designed specifically for constrained nodes and networks, playing a crucial role in the IoT ecosystem. Although the protocol draws inspiration from HTTP by adopting the REST architectural style, CoAP uniquely operates over UDP, distinguishing itself through asynchronous request and response management via CoAP messages. This design results in a simpler protocol, free from historical complexities. CoAP follows a client/server interaction model similar to that of HTTP, but it is specifically optimized for machine-to-machine communications that dominate its use. In these contexts, CoAP devices frequently act as both client and server. A CoAP client begins an interaction by sending a request to a server, using methods such as GET or POST, to act upon a resource identified by a URI on the server. The server responds with either a representation of the resource or a status code indicating the outcome of the action. CoAP accommodates both reliable and unreliable modes of communication. For reliable communication, a CoAP sender transmits a request as a Confirmable (`CON`) message, and the recipient is required to acknowledge receipt by responding with an Acknowledgment (`ACK`) message before a preset timeout period elapses. In cases where the recipient cannot process the `CON` message, it sends back a Reset (`RST`) message as a form of reply. On the other hand, when reliability is not a priority, CoAP allows for the sending of Non-confirmable (`NON`) messages. These messages are sent without the expectation of an acknowledgment.

Figure 1 depicts a typical interaction between two CoAP entities employing a combination of Confirmable and Non-confirmable messages. The interaction begins with Entity A sending a `CON` message to Entity B, requesting the resource located at the "/resource". Entity B responds with an `ACK` message containing the requested resource, signifying a successful GET operation. Subsequently, A issues a `NON` message to perform a POST request, targeting the "/update" URI to modify a resource on B. In reply, entity B sends a `NON` message with status code "2.04 Changed", confirming that the resource has been updated.

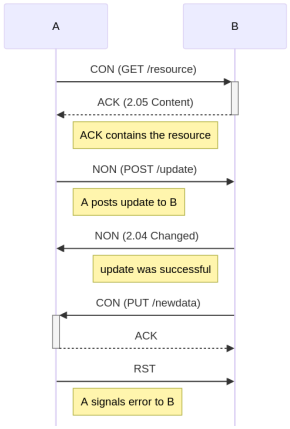

Fig. 1. Sample CoAP Interaction

Entity B then sends a `CON` message with a PUT request to either create or update the resource at the "/newdata". Entity A acknowledges receipt of the PUT request with an `ACK` message. This `ACK` merely indicates the message's receipt and not the success of the PUT operation. The sequence concludes with Entity A sending a `RST` message to Entity B. This message indicates an error encountered during the processing of the PUT request or the inability to fulfill the request, resulting in the termination of the interaction for this specific operation.

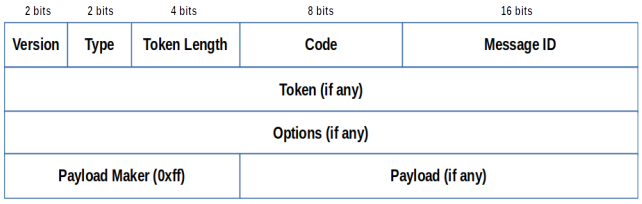

Fig. 2. CoAP Message Format

CoAP employs a concise message format (shown in Fig. 2) for both requests and responses. The format comprises a 4-byte binary header, which may be followed by a series of compact binary options and an optional payload. We will now go over the key fields in a CoAP message.

`Version` The header of a CoAP message begins with a `version` field, represented as a 2-bit unsigned integer.

`Type` The type of a CoAP message is identified by a 2-bit unsigned integer in the message header.

`Token Length` This 4-bit field specifies the length of the `Token` field, which can be between 0 to 8 bytes. The token is used to correlate requests and responses.

`Code` The distinction between a request and a response in CoAP is determined by the code value. This is an 8-bit unsigned integer divided into a 3-bit class (the most significant bits) and a 5-bit detail (the least significant bits), formatted as 'c.dd'. Here, 'c' represents a digit ranging from 0 to 7 for the 3-bit subfield, and 'dd' comprises two digits from 00 to 31 for the 5-bit subfield.

`Message ID` This 16-bit unsigned integer field is used to detect duplicates and, optionally, enable reliable communication.

`Options` A variable-length sequence of options that modify the request or response. Options can specify URI paths, content formats, query strings, and more. Each option is a pair which comprises a number and a value.

`Payload Maker` In CoAP, the presence of a payload in a message is indicated by a specific payload marker [23, p. 17], which is a distinct byte (`0xFF`). This marker serves to separate the payload, which is the actual data being transmitted, from the rest of the message components, such as the header and any options.

With RFC 7959 [5], CoAP incorporates a feature known as *Block-Wise Transfers* to facilitate the transfer of large data. This mechanism breaks down a large body into a series of smaller blocks, which is particularly advantageous in constrained network environments. Such environments often contain devices with limited memory and bandwidth capacities, making the transmission of a large body in a single message impractical. A CoAP entity indicates the need for block-wise transfer through the inclusion of either a `Block1` or `Block2` option within the CoAP message. The `Block1` option is employed in requests, particularly when a client needs to send a large body to the server. Conversely, the `Block2` option is utilized in responses, allowing the server to send a large body back to the client. These options are used in managing both the size and sequence of the blocks within the exchanged messages. The value of a Block option encodes the following fields:

`Block Size (SZX)` Specifies the size of each individual block.

`More Blocks Indicator (M)` Indicates whether additional blocks will follow the current block.

`Block Number (NUM)` Represents the relative number of the current block within a series of blocks of the given size.

IV. Methodology

In this section, we describe our methodology. In a nutshell, it involves creating a common test harness (§IV-A), encoding protocol requirements as assertions within the SUT (§IV-B), and systematically applying fuzzing (§IV-C) or SE (§IV-D) to detect deviations from CoAP protocol requirements.

A. Test Harness Creation

The first step is to create a test harness that interacts with a protocol entity. This harness is responsible for facilitating the testing process by sending and receiving protocol messages. Before testing can start, it is necessary to capture a sequence of packets that are sent to the protocol entity during an interaction (similar to Fig. 1). Packet capture can be achieved by configuring sample programs typically provided with the SUT to perform interactions, and then using tools like TCPdump or Wireshark to capture the packets that are sent to the protocol entity (e.g., server) during one such interaction. During testing, the test harness advances the protocol interaction step by step using pre-captured packets. At each step, it invokes the appropriate API from the SUT to process the packet and generate the next packet from the corresponding response. The test harness should be able to run test scenarios that are relevant for all checked requirements. This can be achieved either by

TABLE I
COAP REQUIREMENTS EXTRACTED FROM RFC 7252 AND RFC 7959

Requirement	Short Description	Reference
Version Validity	The version field MUST be set to 1 (01 binary). Other values are reserved for future versions. Messages with unknown version numbers MUST be silently ignored.	RFC 7252 [23, p. 16]
Matching Message Type	Confirmable messages MUST either carry a request or response, or be empty to elicit a Reset. Non-confirmable messages MUST always carry a request or response. Acknowledgement messages MUST carry a response or be empty. Reset messages MUST be empty. If these conditions are not met, the recipient MUST reject the message.	RFC 7252 [23, p. 21, 23]
Reserved Code	The class can indicate a request (0), a success response (2), a client error response (4), or a server error response (5). All other class values are reserved and MUST be rejected.	RFC 7252 [23, p. 16]
Token Length Validity	Lengths 9-15 for tokens are reserved, MUST NOT be sent, and MUST be processed as a message format error.	RFC 7252 [23, p. 16]
Token Echo	The tokens in a response and its respective request MUST match.	RFC 7252 [23, p. 35]
Message ID Echo	The message ID transmitted in a `CON` or `NON` message, MUST be echoed in the `ACK` or `RST` message by the recipient.	RFC 7252 [23, p. 24]
Repeatable Options	An option that is not repeatable MUST NOT be included more than once in a message.	RFC 7252 [23, p. 39]
Unrecognized Options	Unrecognized options in a message MUST be rejected.	RFC 7252 [23, p. 37]
Block Size Validity	If the block is not the final block, the block size implied by SZX MUST match the size of the payload in bytes.	RFC 7959 [5, p. 10]
Content Format	The Content-Format Option sent with the requests or responses MUST reflect the Content-Format of the entire body.	RFC 7959 [5, p. 12]
Further Request Block Size	The SZX block size specified in a Block1 Option in control usage in a response indicates the maximum block size preferred by the server for subsequent transfers, and the client should adhere to this size or a smaller one in all further requests within the transfer sequence.	RFC 7959 [5, p. 14]
Missing Blocks	If not all previous blocks are available at the server at the time of processing the final block, the transfer fails and error code 4.08 MUST be returned.	RFC 7959 [5, p. 14]

running a single interaction that is relevant for all requirements, or by executing several distinct interactions. In our work, the constructed test harness serves as a unified mechanism for requirements checking, and can be easily adapted either for fuzzing or for symbolic execution with minimal adjustments.

B. Encoding Protocol Requirements as Assertions

Network protocols involve particular requirements for the sequences of packets exchanged among parties. For CoAP, these requirements are located in its RFC documents including RFC 7252 [23] and RFC 7959 [5]. Once requirements, such as those listed in Table I, have been extracted from the RFC documents, they can be translated into assertions over sequences of packets exchanged between protocol entities. The assertions are then incorporated into the SUT code, in the region responsible for sending a response to a received packet. In the two implementations we tested, assertions were inserted into function `coap_session_send_pdu` for libcoap, and function `coap_server_trans_send` for FreeCoAP. In both cases, these two functions already took as argument a pointer to the memory block storing the response. Our assertions used this pointer to access fields in the response packet. Fields in the received packet were accessed via a global pointer variable which was maintained externally in the test harness.

Let us now present the encoding of two example CoAP requirements by quoting the relevant RFC text and providing the corresponding assertion that checks the requirement. The CoAP RFC [23, p. 24] specifies that:

> *The Message ID is a 16-bit unsigned integer that is generated by the sender of a Confirmable or Non-confirmable message and included in the CoAP header. The Message ID MUST be echoed in the Acknowledgement or Reset message by the recipient.*

In Table I, we refer to this requirement as Message ID Echo. Let $\mathtt{p_{in}}$ represent the parsed transmitted packet and $\mathtt{p_{out}}$ represent the response packet. We place the following assertion at the point where the response packet is being generated:

$$\begin{aligned}\texttt{assert}\,(((&\mathtt{p_{in}.type} = \texttt{CON} \vee \mathtt{p_{in}.type} = \texttt{NON}) \wedge \\ &(\mathtt{p_{out}.type} = \texttt{ACK} \vee \mathtt{p_{out}.type} = \texttt{RST})) \implies \\ &\mathtt{p_{in}.message\_id} = \mathtt{p_{out}.message\_id})\end{aligned}$$

During the execution of the test harness, this assertion is evaluated at each step to ensure that for every transmitted packet of type Confirmable (`CON`) or Non-confirmable (`NON`), the `message_id` is correctly echoed by the responder.

Some protocol requirements specify relationships between packets received in a sequence. To capture such relationships, we can formulate constraints over the fields of a sequence of packets received by a protocol party. For instance, RFC 7959 [5, p. 14] mandates:

> *In response to a request with a payload (e.g., a PUT or POST transfer), the block size given in the Block1 Option indicates the block size preference of the server for this resource.*
>
> *...*
>
> *the client SHOULD heed the preference indicated and, for all further blocks, use the block size preferred by the server or a smaller one.*

In Table I, this requirement is referred to as Further Request Block Size. We check this requirement using the excerpt below:

$$\begin{aligned}&\texttt{if}\,(\texttt{first\_response} \wedge \mathtt{p_{in}.code} \in \texttt{REQ\_CODES}) \\ &\quad \texttt{preferred\_size} := \texttt{BlockSize}(\mathtt{p_{out}}) \\ &\quad \texttt{first\_response} := \texttt{false} \\ &\texttt{else} \\ &\quad \texttt{assert}((\texttt{BlockSize}(\mathtt{p_{in}}) > \texttt{preferred\_size}) \implies \\ &\qquad\qquad \mathtt{p_{out}.code} \in \texttt{ERR\_CODES})\end{aligned}$$

In this code, `first_response` is a boolean variable that is `true` if $\mathtt{p_{out}}$ is the first packet generated by the recipient, and `preferred_size` is a variable that stores the server's preferred block size. `BlockSize` is a function that takes a packet and returns the block size given in the Block1 option. Finally, `REQ_CODES` represents the set of codes used for requests, while `ERR_CODES` denotes the set of error codes. The above code stores the preferred size communicated in the first block transmitted by the server. It then checks that subsequent requests that exceed this size are rejected by the server, prompting it to generate an error response. Note that the RFC does not explicitly define a suggested response when a transmitted block fails to meet the preferred size requirement. In cases where the RFC is unclear on the expected behavior, we can use common sense. Here, we require that the response somehow flags an error.

Once the SUT's code has been extended with such assertions, we can employ fuzzing or SE to test whether these assertions are triggered (i.e., whether the requirements can be violated).

C. Testing Requirements Using Fuzzing

To test using fuzzing, we employed AFL++ (4.10c), a state-of-the-art fuzzer [10] incorporating contributions from

TABLE II
RESULTS FROM TESTING THE COAP REQUIREMENTS ON LIBCOAP AND FREECOAP USING FUZZING AND SYMBOLIC EXECUTION

	libcoap (version 4.3.1)						FreeCoAP (commit ffc87fd)					
		Fuzzing		Symbolic Execution				Fuzzing		Symbolic Execution		
Requirement	**Bug Status[1]**	**TTE[2]**	**MTE[3]**	**TTE[2]**	**Time[4]**	**Paths**	**Bug Status[1]**	**TTE[2]**	**MTE[3]**	**TTE[2]**	**Time[4]**	**Paths**
Version Validity	Fixed (#1376)	<1s	193	1s	1s	2	—				43s	6
Matching Message Type	Fixed (#1295)	<1s	371	4s	16s	38	—				2h51m4s	7844
Reserved Code	Fixed (#1300)	<2s	546	4s	17s	41	—				2h55m32s	7844
Token Length Validity	—				12m42s	31	—				3h00m14s	7083
Token Echo	—				1s	2	—				20s	4
Message ID Echo	—				1s	1	—				11s	32
Repeatable Options	Fixed (#1389)	<1s	205	56s	2m29s	213	Reported (#41)	<2s	789	3m30s	🕒	28949
Unrecognized Options	—				2m36s	196	—				🕒	28917
Block Size Validity	Fixed (#1284)	<1s	286	1s	2h25m28s	17971	—				37s	203
Content Format	—				1s	4	—				1s	1
Further Request Block Size	Fixed (#1290)	<2s	891	2s	19m14s	27	Reported (#42)	≈40s	36824	3s	45m13s	8497
Missing Blocks	Fixed (#1287)	<1s	255	6s	3m51s	359	—				1h24m17s	22979

[1] **Bug Status**: non-empty entries report on the bug's status at the time of this writing; in parentheses is the issue number in the GitHub repositories of libcoap and FreeCoAP.
[2] **TTE (Time to Exposure)**: the time it takes to expose the requirement violation (average across five AFL++ fuzzing campaigns or KLEE runs).
[3] **MTE (Mutations to Exposure)**: the number of mutations it takes for AFL++ to expose a requirement violation (average across five fuzzing campaigns).
[4] **Time**: the time required for symbolic execution to complete its exploration, with a timeout set at 24 hours (denoted 🕒).

the latest research. As initial seeds, we utilized valid pre-captured packets obtained from interactions executed using the sample programs provided with each implementation. The fuzzer begins by executing the test harness, which performs a test scenario. It then provides the incoming packet, intended to be processed by a protocol party, by mutating these seeds. For some requirements concerning block-wise transfer, multiple packets need to be processed before an assertion can be checked, as a state needs to be built. Since standard fuzzers cannot handle these stateful scenarios, we load all the required packets for the interaction into a single buffer at different offsets. During a block-wise transfer, for each packet, the recipient reads from the appropriate offsets within the buffer. Consequently, the fuzzer can mutate this single buffer during a fuzzing campaign. All the assertions are checked simultaneously. To account for the inherent randomness of fuzzing, fuzzing campaigns are repeated multiple times. When the fuzzer detects an assertion violation, it stores the mutated test case that caused the violation, along with metadata such as the timestamp and number of mutations before the test case is generated. Analyzing the generated test case can help identify the reason for the assertion violation.

D. Testing Requirements Using Symbolic Execution

To test using symbolic execution, we employed KLEE (v3.0), a mature symbolic execution engine [6] built on top of the LLVM compiler infrastructure. During each symbolic execution run, only one requirement is checked, necessitating separate runs for each requirement. In each run, KLEE executes the test harness, which performs a test scenario using pre-captured valid packets. For each requirement, before a packet is processed by a protocol party, the relevant fields specified in the requirement are treated as symbolic using a KLEE-specific API, while other fields in the packet remain unchanged. To facilitate the process of treating a field as symbolic, we used a CoAP parser to convert loaded packets into data structures. We also used a serializer to convert the data structure back into a buffer, which is then processed by the protocol party. For instance, to check the Message ID Echo requirement, we treated the `message_id` and the `type` in the packet as symbolic. During each SE run, if a path exists where the checked assertion is violated, KLEE, with the help of an SMT solver, generates a concrete test case that violates the assertion. In this context, the test case refers to the specific values of the fields that were made symbolic.

V. EVALUATION

In this section, we present results from applying our methodology to two well-known CoAP implementations: libcoap and FreeCoAP. We begin by comparing the effectiveness of the two techniques (§V-A), followed by a description of the bugs and non-conformances they discovered (§V-B).

A. Effectiveness of Fuzzing and Symbolic Execution

In terms of effectiveness in detecting requirement violations, we did not notice any significant difference between the two techniques. On this set of twelve CoAP requirements, both techniques identified non-conformances for seven of them in libcoap —all of them have been fixed— and for two of them in FreeCoAP which have been reported (cf. Table II). Also, all violations were found quickly: We ran five fuzzing campaigns, and AFL++ consistently found inputs that violate the assertions in less than two seconds and 900 mutations on average, with the exception of one case in FreeCoAP where it needed about 40 seconds and 36 824 mutations on average to find the violation. Symbolic execution using KLEE was also able to find inputs that violate the assertions in few seconds for most requirements, with the exception of the Repeatable Options requirement for which it needed about one minute for libcoap and three and a half for FreeCoAP to find inputs that trigger the violation.

For the protocol requirements that an implementation respects, the two techniques differ in a fundamental way. Fuzzing, being a random testing technique, cannot give any guarantee other than "No violation was found after trying N input mutations in time T." Hence, Table II does not show any numbers for fuzzing for requirements that are not violated; any

such number depends on the time limit T used for the fuzzing campaign. In contrast, symbolic execution systematically explores the program paths and can in principle provide a guarantee that a requirement is not violated. The Time and Paths columns of Table II report the time which KLEE took to explore the corresponding number of paths. For libcoap, we can see that KLEE explored the search space of most requirements in few seconds or minutes with the exception of the Block Size Validity requirement for which it needed to explore about 18 000 paths in two and a half hours. Exploring the search space of FreeCoAP's code proved more challenging: for four of the requirements KLEE took more than an hour to complete its exploration and for two other requirements the exploration did not complete within the 24 hour time limit that we used.

Taking the above into account, our experience is that fuzzing has a slight edge when one is interested in finding specification violations fast, but investing on SE pays off in the long run. In most cases, symbolic execution is able to trigger violations equally fast as fuzzing, and it can also provide guarantees of the absence of bugs in cases where the SMT solver used by the SE engine does not time out. Our overall recommendation to implementors is to adopt the assertion-based methodology to testing protocol requirements we have described in this paper.

B. Bugs Discovered

We describe the non-conformances grouped by the corresponding requirement.

Version Validity: A violation of the Version Validity requirement occurs when a server, upon receiving a message m with an invalid version, responds with a Reset message (`RST`) rather than ignoring m. This deviation from the RFC can cause interoperability issues, increase unnecessary network traffic, and potentially be exploited to fingerprint the server.

Matching Message Type: This requirement is violated when there is a discrepancy between the message `Type` and `Code` fields. Our testing revealed that a libcoap server failed to reject an erroneous message of type `ACK` that carries a request. Instead, it responded with the requested resource in a non-confirmable message. This non-conformance can also be used for fingerprinting or cause other problems.

Reserved Code: Another non-conformance arises when entities do not reject messages containing reserved codes. Accepting messages with these codes can cause interoperability issues among different CoAP implementations.

Repeatable Options: This non-conformance arises when a server cannot properly handle an (erroneous) message m containing more than one instance of an unrepeatable option. In such cases, the server proceeded with a response rather than rejecting m. Failing to satisfy this CoAP requirement can lead to interoperability issues and potential erroneous data interpretation. Both libcoap and FreeCoAP implementations exhibited this non-conformance.

The remaining three non-conformances are related to block-wise transfers.

Block Size Validity: A significant bug was detected in libcoap when there is a mismatch between the block size suggested by the `SZX` value and the actual size of a message's payload. In such scenarios, the CoAP entity processes the message based exclusively on the `SZX` value, disregarding the actual payload size. This situation leads to two different problems:

- If the `SZX` value indicates a block size *larger* than the actual payload, arbitrary data can be stored on the server. Such a situation might cause buffer overwrites, overreads, and introduce various security vulnerabilities due to the processing of unintended data.
- Conversely, if the `SZX` value denotes a block size *smaller* than the actual payload, only a fraction of the payload gets stored on the server. This leads to the partial loss of transmitted data, as part of the payload is discarded.

After reporting the bug that our testing revealed, libcoap's development team addressed it through Pull Request (PR) #1286. Subsequent testing of the SUT with the changes in this PR revealed that the fix was only partial. Specifically, the PR resolved the problem only in scenarios where a server receives a request from a client, not in cases where a client receives a response from a server. The issue was fully resolved by another PR (#1294) which was later merged into libcoap's code base.

Further Request Block Size: In the CoAP block-wise transfer process, once the block size has been negotiated, it is expected that all entities will adhere to this agreed-upon size for the remainder of the data exchange. A non-conformance arises when an entity encounters a message m with a block size that deviates from the one previously established. Instead of ignoring m or reporting an error, the entity adjusted its own block size to align with the block size of m. Both libcoap and FreeCoAP implementations exhibited this non-conformance.

Missing Blocks: A non-conformance in libcoap was detected when a server fails to recognize the absence of one or more blocks prior to receiving the final block in a block-wise transfer. Specifically, the issue occurs under the following circumstances: When a CoAP server receives a block with the 'More' (`M`) bit set to 0, it interprets this as the final block of the transfer. At this point, the server is required to have all preceding blocks available in order to successfully reassemble and process the complete data. However, in certain scenarios, the server does not adhere to this requirement. For instance, if data is partitioned into three blocks and the server receives blocks numbered 0, 2, and 3. In such cases, the server is expected to detect this discrepancy and return an error. Nonetheless, the libcoap server erroneously proceeded without recognizing that a block is missing, failing to return the anticipated error code.

VI. Conclusions

In this paper, we investigated the use of two major testing techniques, fuzzing and symbolic execution, to test IoT protocol implementations for conformance to the requirements in their specifications, and applied them to two widely used implementations of the CoAP protocol. Our results demonstrated the effectiveness of these two techniques when combined with the assertion-based methodology we advocate. Most notably, we have shown that both fuzzing and SE are able to uncover

requirement violations that have remained unnoticed by their developers, and that this can be done quickly.

We hold that our work offers practical insights for developers and researchers in the area of IoT protocol requirement testing. By highlighting the strengths and limitations of fuzzing and symbolic execution, we provide guidance on how to improve the security and reliability testing of IoT networks.

Acknowledgments

We thank Sabor Amini for his initial work on using SE to test CoAP implementations [1]. Although our testing infrastructure differs significantly from his, some of the libcoap bugs of Table II were originally discovered and reported by him.

This research was partially funded by the Swedish Foundation for Strategic Research (SSF) through project aSSIsT and by grants from the Swedish Research Council (Vetenskapsrådet).

References

[1] S. Amini, "Using requirement-driven symbolic execution to test implementations of the CoAP and EDHOC network protocols," Master's thesis, Uppsala University, Department of Information Technology, Sep. 2023.

[2] H. Asadian, P. Fiterău-Broştean, B. Jonsson, and K. Sagonas, "Applying symbolic execution to test implementations of a network protocol against its specification," in *IEEE Conference on Software Testing, Verification and Validation*, ser. ICST 2022. IEEE, Apr. 2022, pp. 70–81. [Online]. Available: https://ieeexplore.ieee.org/document/9787883

[3] ——, "Monitor-based testing of network protocol implementations using symbolic execution," in *Proceedings of the 19th International Conference on Availability, Reliability and Security*, ser. ARES '24. ACM, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3664476.3664521

[4] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in *31st USENIX Security Symposium*, ser. USENIX Security 2022. USENIX Association, Aug. 2022, pp. 3255–3272. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/ba

[5] C. Bormann and Z. Shelby, "Block-Wise Transfers in the Constrained Application Protocol (CoAP)," RFC 7959, Aug. 2016. [Online]. Available: https://www.rfc-editor.org/info/rfc7959

[6] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[7] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013. [Online]. Available: https://doi.org/10.1145/2408776.2408795

[8] M. M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler, "Heartbleed 101," *IEEE Secur. Priv.*, vol. 12, no. 4, pp. 63–67, 2014. [Online]. Available: https://doi.org/10.1109/MSP.2014.66

[9] K. Cullen, "Freecoap." [Online]. Available: https://github.com/keith-cullen/FreeCoAP

[10] A. Fioraldi, D. C. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies*, ser. WOOT 2020. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[11] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. [Online]. Available: https://doi.org/10.1145/360248.360252

[12] "libcoap." [Online]. Available: https://www.libcoap.net

[13] F. Liljedahl, "Exploring the possibilities of robustness testing of CoAP implementations using evolutionary fuzzing," Master thesis, KTH, Stockholm, Sweden, 2019.

[14] B. Melo and P. Geus, "Robustness testing of CoAP server-side implementations through black-box fuzzing techniques," in *Proceedings of the 17th Brazilian Symposium on Information and Computational Systems Security*. SBC, 2017, pp. 533–540. [Online]. Available: https://sol.sbc.org.br/index.php/sbseg/article/view/19528

[15] R. Meng, Z. Dong, J. Li, I. Beschastnikh, and A. Roychoudhury, "Linear-time temporal logic guided greybox fuzzing," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE' 22. New York, NY, USA: ACM, May 2022, pp. 1343–1355. [Online]. Available: https://doi.acm.com/10.1145/3510003.3510082

[16] B. Möller, T. Duong, and K. Kotowicz, "This POODLE bites: exploiting the SSL 3.0 fallback," 2014. [Online]. Available: https://www.openssl.org/~bodo/ssl-poodle.pdf

[17] R. Natella, "StateAFL: Greybox fuzzing for stateful network servers," *Empir Software Eng*, vol. 27, no. 7, p. 191, 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10233-3

[18] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. D. Millstein, "Analyzing protocol implementations for interoperability," in *12th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI 15. USENIX Association, May 2015, pp. 485–498. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pedrosa

[19] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: A greybox fuzzer for network protocols," in *IEEE 13th International Conference on Software Testing, Validation and Verification*, ser. ICST 2020. IEEE, Oct. 2020, pp. 460–465. [Online]. Available: https://ieeexplore.ieee.org/document/9159093

[20] C. Poncelet, K. Sagonas, and N. Tsiftes, "So many fuzzers, so little time*: Experience from evaluating fuzzers on the Contiki-NG network (hay)stack," in *37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: ACM, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556946

[21] F. Rath, D. Schemmel, and K. Wehrle, "Interoperability-guided testing of QUIC implementations using symbolic execution," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ@CoNEXT 2018. ACM, Dec. 2018, pp. 15–21. [Online]. Available: https://doi.org/10.1145/3284850.3284853

[22] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, "KleeNet: Discovering insidious interaction bugs in wireless sensor networks before deployment," in *Proceedings of the 9th International Conference on Information Processing in Sensor Networks*, ser. IPSN 2010. ACM, Apr. 2010, pp. 186–196. [Online]. Available: https://doi.org/10.1145/1791212.1791235

[23] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," RFC 7252, Jun. 2014. [Online]. Available: https://www.rfc-editor.org/info/rfc7252

[24] J. Song, C. Cadar, and P. R. Pietzuch, "SymbexNet: Testing network protocol implementations with symbolic execution and rule-based specifications," *IEEE Trans. Software Eng.*, vol. 40, no. 7, pp. 695–709, 2014. [Online]. Available: https://doi.org/10.1109/TSE.2014.2323977

[25] S. Tempel, V. Herdt, and R. Drechsler, "Specification-based symbolic execution for stateful network protocol implementations in the IoT," *IEEE Internet of Things Journal*, 2023. [Online]. Available: https://doi.org/10.1109/JIOT.2023.3236694

[26] S. Wen, Q. Meng, C. Feng, and C. Tang, "A model-guided symbolic execution approach for network protocol implementations and vulnerability detection," *PloS one*, vol. 12, no. 11, p. e0188229, 2017. [Online]. Available: https://doi.org/10.1371/journal.pone.0188229

[27] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2013.

[28] Y. Zeng, M. Lin, S. Guo, Y. Shen, T. Cui, T. Wu, Q. Zheng, and Q. Wang, "MultiFuzz: A coverage-based multiparty-protocol fuzzer for IoT publish/subscribe protocols," *Sensors*, vol. 20, no. 18, 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/18/5194