Hallucinating Certificates: Differential Testing of TLS Certificate Validation Using Generative Language Models

Muhammad Talha Paracha
Ruhr University Bochum
Bochum, Germany
mtalhapar@gmail.com

Kyle Posluns
Northeastern University
Boston, United States
posluns.k@northeastern.edu

Kevin Borgolte
Ruhr University Bochum
Bochum, Germany
kevin.borgolte@rub.de

Martina Lindorfer
TU Wien
Vienna, Austria
martina@seclab.wien

David Choffnes
Northeastern University
Boston, United States
choffnes@ccs.neu.edu

Abstract

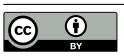
Certificate validation is a crucial step in Transport Layer Security (TLS), the de facto standard network security protocol. Prior research has shown that differentially testing TLS implementations with synthetic certificates can reveal critical security issues, such as accidentally accepting untrusted certificates. Leveraging known techniques, like random input mutations and program coverage guidance, prior work created corpora of synthetic certificates. By testing the certificates with multiple TLS libraries and comparing the validation outcomes, they discovered new bugs. However, they cannot generate the corresponding inputs efficiently, or they require to model the programs and their inputs in ways that scale poorly.

In this paper, we introduce a new approach, MLCerts, to generate synthetic certificates for differential testing that leverages generative language models to more extensively test software implementations. Recently, these models have become (in)famous for their applications in generating content, writing code, and conversing with users, as well as for "hallucinating" syntactically correct yet semantically nonsensical output. In this paper, we provide and leverage two novel insights: (a) TLS certificates can be expressed in natural-like language, namely in the X.509 standard that aids human readability, and (b) differential testing can benefit from hallucinated malformed test cases.

Using our approach MLCerts, we find *significantly* more *distinct discrepancies* between the five TLS implementations OpenSSL, LibreSSL, GnuTLS, MbedTLS, and MatrixSSL than the state-of-the-art benchmark Transcert (+30%; 20 vs 26, out of a maximum possible of 30) and an order of magnitude more than the seminal work Frankencerts (+1,200%; 2 vs 26). Finally, we show that the diversity of MLCerts-generated certificates reveals a range of previously unobserved and interesting behavior with security implications.

CCS Concepts

• **Software and its engineering → Software testing and debugging**; • **Security and privacy → Security protocols**.



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3773203

Keywords

TLS, X.509, TLS Certificate Validation, Differential Testing, Large Language Models, Software Testing, Fuzzing

ACM Reference Format:
Muhammad Talha Paracha, Kyle Posluns, Kevin Borgolte, Martina Lindorfer, and David Choffnes. 2026. Hallucinating Certificates: Differential Testing of TLS Certificate Validation Using Generative Language Models. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773203

1 Introduction

Transport Layer Security (TLS) is the de facto standard network security protocol, providing connection confidentiality, integrity, and authenticity. It is widely used for web, mobile, and Internet of Things (IoT) communication. TLS has undergone rigorous development and extensive testing over the last 25 years. Specification documents (specs) written in natural language, typically English, define TLS's *intended* behavior. For example, the X.509 standard specifies the format of certificates [1]. However, the specs are so complex that it is challenging for developers to implement the protocol from scratch. Implementing your own security protocol is also heavily discouraged, because it can introduce critical vulnerabilities. Instead, developers typically integrate well-tested TLS implementations, like OpenSSL, MatrixSSL, or MbedTLS. As is the case with any software, ensuring that these TLS implementations truly follow the *current* spec correctly and are also free of bugs is difficult. Indeed, bugs and security issues are still regularly discovered in them, from denial-of-service vulnerabilities [2, 3] to leaks of user-sensitive data [4, 5] to improper server authentications [6].

An essential step in the TLS protocol to ensure server authenticity is certificate validation and prior work has focused on systematically finding corresponding bugs in TLS implementations [7, 8, 9, 10, 11]. Considering the lack of ground truth about which TLS implementations are correct, a popular approach, introduced by Frankencerts [7] in the context of TLS, is differential testing with the various implementations acting as cross-referencing oracles. Given an input certificate, differential testing compares the validation outcomes of multiple TLS libraries to identify *discrepancies* between them, that is, certificates that one or more implementations accept but at least one other implementation rejects. Although there can be many reasons for differences in behavior across implementations,

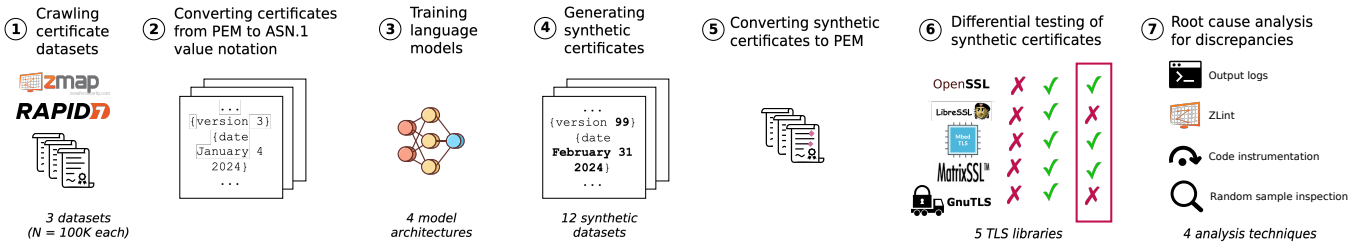


Figure 1: An overview of the methodology used in our work.

prior work has shown that these discrepancies generally indicate implementation bugs and vulnerabilities [7, 8, 12].

The success of differential testing heavily depends on the generation of diverse inputs (in our case, TLS certificates) that comprehensively exercise the implementations' functionality and that have a high chance of triggering output variations. Prior work combined and mutated parts of real certificates [7], used code coverage to guide generation [8], and modeled certificate parameters for combinatorial methods [9]. We propose a new approach to generate diverse input datasets for differential testing, by leveraging generative language models. Recently, these models have become infamous for "hallucinating," that is, they generate syntactically correct but semantically nonsensical or even misleading output (e.g., "there is one m in the word weather"). In our approach, we leverage and give purpose to this flaw of generative language models as a feature, namely to generate a large and diverse set of malformed but syntactically correct (i.e., parseable by the TLS implementations) synthetic certificates for use in differential testing.

Our insight is that we can train generative language models to (i) learn a *representation* for the textual data they are trained on, which can then be sampled to obtain synthetic instances, (ii) learn a representation that is probabilistic and often *imperfect*, meaning a sampled instance can considerably break expectations, and (iii) can learn *multiple* imperfect representations depending on the model architecture and training dataset. For the differential testing of TLS libraries, we (i) remove the need to manually create a grammar of TLS certificates to use for generating test datasets, (ii) show that an imperfect representation can actually improve testing, as software bugs are often found through malformed inputs that trigger rarely executed code paths, and (iii) find that multiple representations can help produce test instances that differ significantly in their features, and thus trigger different code paths during testing.

In this paper, we make the following contributions:

- We introduce MLCerts, a new approach that leverages generative language models to generate diverse X.509 TLS certificates.

  Notably, while large generative language models are popular today (e.g., GPTs), we show the surprising effectiveness of training classical language models (e.g., RNNs) that have been available for over two decades and require substantially fewer resources. Our results, in turn, encourage the use of these simpler language models in other software testing domains.

- We differentially test five major TLS libraries, namely OpenSSL, LibreSSL, GnuTLS, MbedTLS, and MatrixSSL, utilizing a diverse dataset of 1M MLCerts-generated synthetic certificates.

  Our MLCerts-generated certificates test the TLS implementations more extensively and produce significantly more *distinct discrepancies* than the state-of-the-art benchmark Transcert (20 vs. 26; out of a maximum possible of 30) and an order of magnitude more than the seminal work Frankencerts (2 vs. 26).

- We introduce novel semi-automated methods to analyze and better understand the root causes of discrepancies.

  We observe a wide range of previously unobserved and interesting behavior of TLS libraries with security implications, such as accepting certificates with invalid dates (e.g., February 31) or badly formatted email addresses. We further find two behaviors to be closely linked with recently fixed bugs: A minor one in GnuTLS, and, a CVE of moderate severity in OpenSSL.

Artifacts: To ensure reproducibility and to enable future research, our datasets of synthetic certificates, code for training the models, and to perform our analysis, are publicly available [13, 14].

2 Motivation & Background

Following, we provide the background for TLS and differential testing, how it relates to language models, and how we can utilize them for generating (wrong) test inputs.

2.1 TLS Protocol and Certificates

A crucial step of proper TLS connection establishment is the validation of one or more certificates' cryptographic information to ensure host authenticity (e.g., that the connection is to *google.com* and not an imposter). TLS clients must validate these certificates according to the protocol spec that, among numerous other requirements, requires the client to determine if a certificate is authentic based on if it is signed by a trusted Certificate Authority (CA), not expired at the time of check, and indeed issued for the host name that the client wants to connect to. Over the past two decades, the TLS spec has become complex and is semi-formally defined in various RFCs [15, 16]. TLS certificates are defined in a different standard, known as X.509 [1], which is based on the interface description language ASN.1 [17]. ASN.1 formats can be efficiently serialized and deserialized across platforms. TLS certificates typically use the binary Distinguished Encoding Rules (DER) format [18] and are encoded as Privacy-Enhanced Mail (PEM) [19]. X.509 has also undergone changes, with version 3 (v3) [20] being the latest.

TLS implementations, like OpenSSL, LibreSSL, or GnuTLS, aim to conform to the spec, but due to TLS' sheer complexity, development mistakes can happen easily and lead to vulnerabilities [2, 4, 6].

```
toBeSigned {
  version 2 -- v3 --,
  serialNumber 104350513648249232941998508985834464573,
  signature {
    algorithm {1 2 840 113549 1 1 5}
  },
  issuer rdnSequence : { ... },
  validity {
    notBefore utcTime : "061201000000Z" -- Fri Dec  1 00:00:00 2006 --,
    notAfter utcTime : "291231235959Z" -- Mon Dec 31 23:59:59 2029 --
  },
  extensions {
    {
      extnId {2 5 29 14} -- id-ce-subjectKeyIdentifier --,
      extnValue '04140B58E58BC64C1537A440A930A921BE47'H
    }
  },
  ...
}
```

Listing 1: A TLS certificate in ASN.1 value notation.

2.2 Differential Testing of Certificate Validation

Numerous approaches assess the TLS protocol and its implementations, as Swierzy et al. have recently surveyed [21] and which we discuss later (see Section 7). Here, we focus on Frankencerts by Brubaker et al. [7], which leverages differential testing to assess TLS certificate validation. Brubaker et al. identified two requirements for systematically testing TLS implementations' certificate validation logic: (i) being able to generate a diverse set of syntactically correct X.509 certificates that exercise rarely triggered code paths, and (ii) being able to determine if a certificate validation result given by an implementation is correct. They proposed a new approach that meets these requirements by (i) randomly combining and mutating parts of real certificates to generate a large synthetic corpus of certificates (Frankencerts) that are syntactically valid but may violate semantic constraints, and (ii) using multiple TLS libraries as cross-referencing oracles. For the cross-reference oracle, they leverage the idea that all TLS libraries attempt to conform to the same protocol spec and should behave the same. This implies that a *discrepancy*, that is, when at least one library accepts a certificate but at least one other library rejects it, can indicate a potential bug in any of the libraries. They observed that the majority of discrepancies that they found are bugs and security issues in the TLS implementations. Other work later also uses this testing technique (see Section 7), with Transcert [11] being the state of the art for differential testing of TLS certificate validation logic.

2.3 Security Testing and Language Models

We hypothesize that generative language models can improve differential testing for programs that need diverse textual inputs. One of our key insights is that we can adapt these models to TLS certificates because their X.509 format can be deserialized from a memory-efficient DER/PEM format into a human-readable format (see Listing 1), a context where language models perform well. We can train a model on a corpus of TLS certificates, that is, learn a representation of them, and then use it to generate new synthetic certificates by sampling from the representation. Interestingly, these models learn representations that are not perfect, resulting in "hallucinated" outputs that differ from expectations. For differential testing, these imperfections can actually help us find bugs. They can deviate slightly from the TLS specification, which can trigger rarely tested code paths and increase the chance of finding bugs.

Language models have not yet been used to assess TLS certificate validation or to generate synthetic TLS certificates, but have shown promise in other domains. Godefroid et al. [22] used language models based on neural networks to generate complex PDF files to then fuzz the PDF parser of Microsoft Edge. While they successfully learned a representation that could generate novel PDF objects, they only identified one stack-overflow bug. Notably, they also did not rely on differential testing, but they only analyzed one target and used AppVerifier, which is a run-time monitoring tool to catch memory bugs. Nevertheless, they identified various promising aspects of the technique and they provided an early initial discussion on how the model-training process influences the quality of fuzzing (see Section 4 for more details from our own investigation). In this paper, we advance their initial idea, adapt it from PDF files to TLS certificates, complement the use of language models with differential testing, and show that our approach can outperform the traditional state of the art.

2.4 Generative Language Models

Language models have received substantial attention by the research community and a plethora of approaches exist. Here, we provide the necessary background to leverage them as part of MLCerts for differential testing. In plain terms, modern generative language models based on neural networks learn to predict the next token(s) when given an input sequence. A token can be a single character, multiple characters, or a unit in another domain than natural language (e.g., a musical note). This makes them extremely powerful and enables their use in numerous contexts.

We use two types of generative language models: Generative Pre-trained Transformers (GPT) and Recurrent Neural Networks (RNNs). GPTs, which rely on a Transformer architecture [23], are the state of the art, while RNNs like the Long Short-Term Memory (LSTM) variant from 1997 [24] were the dominant sequence prediction models in the previous two decades with widespread impact [25]. GPTs are considered Large Language Models (LLMs), due to their scale of trainable model parameters, datasets, and needed compute requirements, while RNNs are not LLMs. For both RNNs and GPTs, the output tokens are sampled from a probability distribution, that is, the models do not output a single token directly, but they output a score for each token as a probability distribution. The output sampling strategy then determines how "creative" or constrained the outputs from a generative model are allowed to be. It is common to select the token with the highest probability as the output token (we provide an empirical analysis on using other strategies in Section 5.2).

Language models are typically trained on a corpus of textual data. It is common to use a pre-trained GPT model, that is, a model that has been pre-trained on a (very) large corpus of generic data with extensive compute, and then to only fine-tune it with a small set of data tailored to the task of the new model (instead of pre-training the GPT model architecture from scratch). For MLCerts, we leverage RNN and GPT models that we train from scratch, and a pre-trained GPT model that we fine-tune. We aim to understand how such off-the-shelf models can improve differential testing, without needing custom model architectures tailored to software testing (while interesting, this is beyond the scope of our work).

Table 1: Overview of certificate datasets used for training language models. * denotes median values.

		Versions			Extensions			Validity (Year)	
Dataset	**Total Certificates**	v1	v3	Others	Per PEM Certificate*	Unique Names	Unique Values	notBefore*	notAfter*
IPv4	100,000	1,790	98,207	3	8	28	92,630	2019	2025
Modern	100,000	11,893	88,102	5	9	31	110,916	2021	2022
Balanced	100,000	50,036	49,944	20	0	29	85,480	2020	2023

3 Goals

We aim to answer the following research questions (RQs):

RQ1: How can we leverage generative language models to precisely generate synthetic X.509 TLS certificates?

RQ2: How can we generate sufficiently diverse certificates for testing TLS implementations, that is, certificates that require exercising different parts of the libraries?

RQ3: Can we uncover new bugs, discrepancies, or security issues through differential testing using our generated set of diverse certificates, and how can we better (automatically) understand the issues we discovered?

4 Approach

MLCerts combines synthetic certificate generation, via generative language models, with differential testing (see Figure 1). We discuss the following steps of our approach in more detail: collecting certificate datasets (step 1), converting them to ASN.1 (step 2), training language models (step 3), generating synthetic certificates (step 4) and converting them to PEM format (step 5), as well as designing our differential testing framework (used in step 6).

4.1 Certificate Datasets

Our approach relies on an input set of TLS certificates to train the language models. The models, in turn, learn the representation of a certificate based on the features of the training data. For instance, if we train the models using only X.509 *v1* TLS certificates, then the models will be unlikely to generate certificates with TLS extensions that are only available in X.509 *v3* TLS certificates. Thus, to generate synthetic certificates with a diverse range of features, we use three distinct datasets (see Table 1):

IPv4 (n = 100,000): We collect certificates by randomly crawling the IPv4 space for TLS-enabled hosts using Zmap [26] until we obtain 100,000 certificates. This dataset represents certificates that are typically found online.

Modern (n = 100,000): We use a dataset of 100,000 certificates observed by Rapid7 scans [27] that were deployed in 2022. This dataset only includes certificates that were first observed in 2022 and excludes those that Rapid7 encountered in their scans before. Thus, the certificates are relatively recent in comparison to IPv4 and include modern features and extensions.

Balanced (n = 100,000): We create a balanced dataset of *v3* and *v1* certificates sampled from the Rapid7 scans, to ensure that *v3* and *v1* certs are equally represented.

Table 1 provides an overview of our datasets and the important certificate features (i.e., versions, extensions, and validity period), and it illustrates the diversity of our datasets.

Dataset Postprocessing. We decode raw TLS certificates into the ASN.1 textual format using PyCrate [28]. This textual format is more suitable for language models (see Listing 1) because of its context and verbosity. To facilitate model training, we remove the public key and signature fields from the training certificates. These special fields need to be cryptographically computed and the language models cannot learn these functions. If we would generate certificates with invalid signatures, then these certificates would be of limited value because TLS validation logic would fail early, preventing us from identifying bugs. Thus, we do not consider these fields for certificate generation by the language model, but we later process the output certificates to compute these fields.

4.2 Language Models

Following, we discuss how we train the language models and the concrete details on how to generate synthetic certificates.

4.2.1 Parameter Selection. Training modern machine learning models is parameterized. Their performance depends not only on the training dataset and model architecture, but also on hyperparameters, like the learning rate, number of layers, batch size, etc. When training them, the learning goal is expressed through the loss function used for optimization. The parameters are empirically selected to get a model with minimum local loss, using a partial search of the vast parameter space for a fixed amount of time.

Unfortunately, this approach does not fit software testing well. Models optimized for the default loss functions will generate outputs that faithfully represent the training data, while testing benefits from inputs that violate expectations, up to a certain margin, to trigger previously unseen code paths instead of rejecting an invalid input early. Indeed, Godefroid et al. [22] highlighted this inherent tension between learning and fuzzing. Learning optimizes for generating outputs that conform to the training data, while fuzzing exploits inputs that break the structure to reach bug-triggering code paths. How one would design a loss function tailored to discovering bugs is unknown though, as it heavily depends on the input format and the set and classes of bugs or vulnerabilities one wants to find.

Considering this inherent tension, it is unknown whether a pre-trained state-of-the-art model (e.g., a GPT) or a simpler model (e.g., a RNN) would be better for differential testing. We aim to answer how to best train and tune generative models for testing using the following approach: First, we train 12 different models based on three training datasets and four model architectures using established parameters (e.g., a moderate learning rate). Second, we evaluate how the different models help us answer our RQs.

Table 2: Overview of synthetic PEM certificates. Each model generated 100,000 ASN.1 outputs. * denotes median values.

			Versions			Extensions			Validity (Year)	
Training Data	**Model**	**PEM Certificates**	v1	v3	Others	Per PEM Certificate*	Unique Names	Unique Values	notBefore*	notAfter*
IPv4	RNN-Small	35,310	678	34,631	1	7	24	43,397	2020	2024
	RNN-Medium	97,151	1,897	95,252	2	7	35	107,105	2020	2025
	GPT-Finetuned	82,081	1,258	80,818	5	7	37	33,010	2016	2027
	GPT	74,431	369	74,062	0	7	21	25,719	2015	2027
Modern	RNN-Small	69,953	20,650	49,301	2	3	18	106,509	2021	2022
	RNN-Medium	34,886	7,304	27,580	2	3	19	46,710	2021	2024
	GPT-Finetuned	74,215	23,356	50,847	12	1	42	71,693	2021	2024
	GPT	44,468	12,567	31,897	4	3	19	64,566	2021	2024
Balanced	RNN-Small	938	711	227	0	0	12	675	2020	2023
	RNN-Medium	95,965	65,833	30,130	2	0	27	60,235	2020	2023
	GPT-Finetuned	81,295	61,581	19,710	4	0	39	28,557	2020	2023
	GPT	64,254	47,050	17,204	0	0	17	35,101	2020	2023

4.2.2 Model Details. Overall, we train four model architectures, with sizes that differ by up to two orders of magnitude, on our three datasets, for a total of 12 models. During our training, we convert each PEM certificate into its textual ASN.1 value notation (step 2) and we add a prefix "BEGINCERTIFICATE" and a suffix "ENDCERTIFICATE". We train our models end-to-end on each certificate, that is, during training, the model sees the entire certificate.

For RNNs, we use the open-source *Char-RNN-PyTorch* implementation [29] with an LSTM variant, akin to prior work [22]. We train two 3-layer models from scratch, one with 256 hidden neurons per layer (*RNN-Small*, with ≈1 million trainable parameters) and the other with 1024 hidden neurons (*RNN-Medium*, with ≈10 million trainable parameters). We treat each character as a token.

For GPTs, we use the open-source *gpt-neo-125m* [30] implementation, which is based on the GPT-3 architecture but has ≈125 million parameters rather than the ≈1 billion of the base GPT-3 model for easier training. Interestingly, while larger models typically lead to better performance, our results did not indicate a need to train larger models (see Section 5). We fine-tune one pre-trained model (GPT-Finetuned) and train another from scratch (GPT) using our datasets. For both models, we set the maximum sequence length to 2048 tokens, the default for GPT-Neo.

4.2.3 Generating Synthetic Certificates. We use the prefix as input to a trained model and try to generate an output string that ends with the suffix, but we stop generation at a maximum output length limit in case the suffix is not output until then. Our models are trained on ASN.1 inputs and we expect the outputs to also be in the ASN.1 language. We then convert the generated certificate from its ASN.1 representation into the PEM format, which TLS libraries can process. The generated certificate output may not actually be well-formed ASN.1 that is compatible with the X.509 format, that is, the conversion is not guaranteed to succeed. We call each certificate that we successfully convert to the PEM format a *synthetic certificate*. We use each synthetic certificate in three ways during differential testing:

Leaf: We use the synthetic certificate directly with a dummy public key and dummy signature.

Leaf+CA: We use a manually trusted v3 root CA as the issuer CA. We include its valid signature and details in the *issuer* field of the synthetic leaf.

Leaf+CA+Intermediate: We chain multiple synthetic certificates to produce a valid certificate chain with one manually trusted root, one leaf, and $n \geq 1$ intermediate certificates.

We form the chain using two random certificates from the synthetic ASN.1 outputs, with one acting as intermediate and other as leaf. We create the certificate chain normally: We use a manually trusted v3 CA to sign the intermediate and then sign the leaf with the intermediate (i.e., we set the *issuer* field accordingly and include a valid signature).

4.3 Differential Testing Framework

After we generated the synthetic certificates, we use them to differentially test TLS libraries. That is, we take a certificate and attempt to validate it using multiple different TLS libraries, and we then compare their validation outcomes against each other, to find *discrepancy-producing certificates*. We categorize the certificate validation outcome for each library as success (✓) or failure (✗). A discrepancy occurs when at least one library differs in validation outcome from the others. For example, one library might accept a certificate, while the others reject it, or vice-versa. We test five popular open-source TLS libraries in their recently-released versions: OpenSSL (3.3.2), LibreSSL (3.9.2), GnuTLS (3.7.11), MbedTLS (3.6.1), and MatrixSSL (4.6.0). Our approach can also easily be extended to support other libraries. OpenSSL, GnuTLS and MatrixSSL were also used in Frankencerts [7]. MatrixSSL is no longer maintained, but it remains widely used. We added MbedTLS as it is targeted towards embedded devices, and LibreSSL as it is a widely used fork of OpenSSL. With five libraries, we can discover a maximum possible of 30 *distinct (sets of) discrepancies*: $2^5 = 32$ outcomes, minus two for all success/failure outcomes.

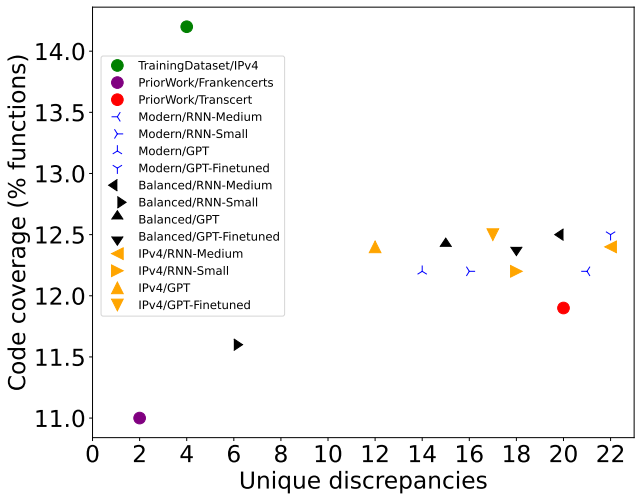


Figure 2: Performance of models in triggering distinct discrepancies and overall code coverage. Our models trigger the most discrepancies, comparable only to Transcert (●), despite covering less code than real-world certificates (●).

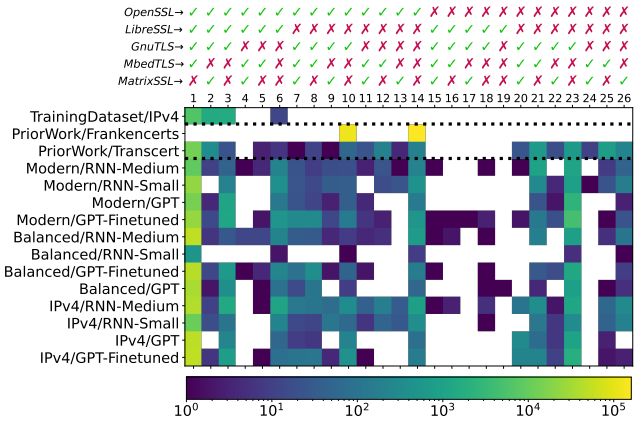


Figure 3: Exploring the discrepancy-producing certificates. Each cell shows the number of certificates per model (row) that trigger a particular discrepancy (column). Rows 1–3 represent benchmarks, and rows 4 and onward represent certificates generated by our MLCerts models. The synthetic certificates trigger significantly more discrepancies than the benchmarks, and mostly overlap on the type of discrepancies.

During testing, we also collect outputs to `stdout` and `stderr` for each certificate alongside aggregate code coverage data using gcov [31] for all certificates of a dataset. When validating certificates, we do not perform domain name validation, as the certificate generation process is not constrained to a target domain.

Certificate validity periods, that is, whether a certificate has expired or is still valid at the time of testing can significantly affect code execution during differential testing (and our results). If a certificate has expired or is not yet valid, then there is no reason for the libraries to check specification compliance for other parts of the certificate, enabling them to fail early without providing any new insight, even if the certificate would have caused a discrepancy. A certificate that is valid for the current time will likely lead to more code being executed, even if not necessarily code paths that are rarely triggered and that may expose subtle bugs. However, given that there is no *a priori* way to determine the optimal date (there may also not be any optimal date), we aim to explore more code paths. Therefore, we validate all certificates using the same date and time, which we do by using libfaketime [32] to patch the used low-level system libraries to return a fixed time for our experiments. We fix the date to June 15, 2022 because the majority of certificates from our training dataset (see Table 1) and of our generated synthetic certificates (≈80%) are then valid during testing.

4.3.1 ZLint. We use ZLint [33] to evaluate the correctness of TLS libraries in their certificate validation outcomes. ZLint is a widely used X.509 certificate linter that checks if a certificate conforms to protocol and other specifications (e.g., with RFC 5280 [20] or the CA/Browser Forum Baseline Requirements [34]). ZLint's checks are composed of "lints," which have descriptions denoting their purpose and a reference to the corresponding rule of the specification. For example, the lint *e_ext_aia_marked_critical* checks that the Authority Information Access (AIA) extension in a certificate is marked non-critical, as RFC 5280 Section 4.2.2.1 requires it [20].

5 Evaluation

We evaluate MLCerts in three parts. First, we compare our trained language models with respect to code coverage and unique discrepancies for the synthetic certificates generated with them (Section 5.1). Second, we investigate how certificate diversity influences the discovery of discrepancies (Section 5.2). Last, we examine the security impact of the discrepancies we discovered (Section 5.3).

We generate 100,000 X.509 TLS certificates in the ASN.1 textual format with each trained model. Some outputs might not be valid ASN.1 and we might be unable to convert them to PEM certificates. Table 2 shows an overview of the successfully generated PEM certificates and their features (RQ1). The models successfully generate valid PEM certificates, with up to 97.15% valid PEM certificates for IPv4/RNN-Medium. The only exception is the Balanced/RNN-Small, as only 0.94% ASN.1 outputs could be converted to PEM certificates.

The different models learn distinctive X.509 features in certificate versions and their extensions. The models trained on the IPv4 and Modern datasets produce more v3 certificates than the models trained on the Balanced dataset. The Balanced models also learned that v1 certificates do not have extensions, their median number of extensions is zero. Notably, the models not only learned X.509 format constraints, but they also learned other constraints from the real-world certificates of our training data. For example, for all models, the certificates' median notAfter field (expiry date) is ahead of the notBefore field (typically the certificate issuance date).

5.1 Discrepancies and Code Coverage

We find no discrepancies when directly using the synthetic certificates with the TLS libraries in Leaf mode. This is expected, because all libraries mark certificates as invalid that are not signed by a valid CA. With a valid chain, in Leaf+CA or Leaf+CA+Intermediate mode (see Section 4.2.3), we find 26 and 19 distinct discrepancies

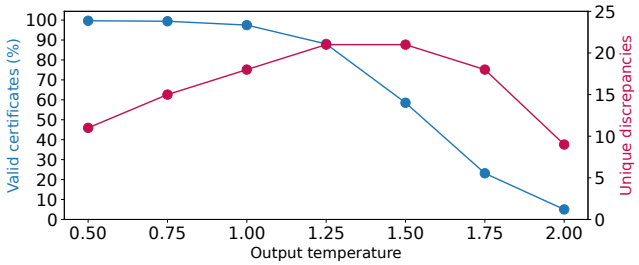


Figure 4: Trade-off between sampling strategy and discrepancies found. Sampling in a conservative way (left) makes more synthetic certificates valid, but they find less discrepancies.

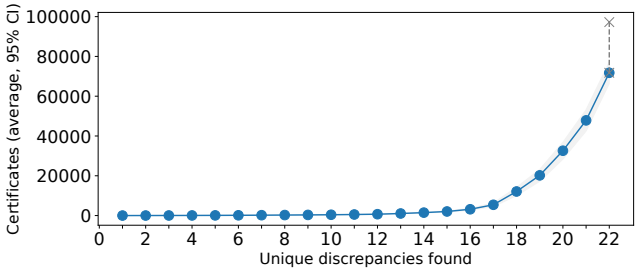


Figure 5: The number of y certificates required to observe x discrepancies with the model IPv4/RNN-Medium. Most unique discrepancies are discovered with just 20,000 synthetic certificates. The gray dashed arrow shows certificates that did not lead to the discovery of new discrepancies.

respectively, with the 19 Leaf+CA+Intermediate discrepancies being a subset of the 26 unique discrepancies. Therefore, for the sake of simplicity, we focus our analysis on the Leaf+CA mode.

Figure 2 shows the performance of the models in unique discrepancies (x-axis) and code covered (y-axis). We compute code coverage as the median percentage of executed functions among all libraries for a dataset. In addition to our 12 models, we include three baselines: our IPv4 training dataset and certificates that we generated with the best-performing Frankencerts [7] and Transcert [11] approaches (see also our artifact [13, 14]).

Most of our MLCerts models trigger a significant number of discrepancies (min 6, max 22, median 17.5). Only Transcert (🔴) is comparable (20) and we widely outperform Frankencerts (🟣) and the IPv4 dataset (🟢). However, the IPv4 training dataset achieves the highest code coverage. Code coverage appears low, but this is because only a subset of the libraries' code is used for certificate validation. We suspect that the real-world certificates have higher code coverage because of outliers. An outlier certificate in the IPv4 dataset will lead to higher code coverage, but it is unlikely that our models reproduce a similar outlier by nature of it being an outlier. Moreover, our models learn a compressed representation to discover discrepancies, not to achieve higher code coverage. Indeed, as Figure 2 clearly shows, higher code coverage does not imply discovering more discrepancies. On the contrary, MLCerts discovers significantly more discrepancies at lower code coverage.

Interestingly, none of our models or their underlying training dataset dominate code coverage or unique discrepancies over the others. The models that trigger large numbers of discrepancies have different architectures and were trained on different datasets. For example, our model IPv4/RNN-Medium triggers the same number of discrepancies (22) as our model Modern/GPT-Finetuned.

Figure 3 visualizes all unique discrepancies across all models and the number of certificates that trigger each type of discrepancy as a heat map. Across all, we discovered 26 unique discrepancies out of the theoretical maximum of 30 discrepancies (see Section 4.3). No individual model finds all of them. In comparison to MLCerts, Transcert misses six discrepancies (4 and 15–19). Transcert cannot find these discrepancies because of limitations of its mutation operators (see Section 6). Three models found one discrepancy each that no other model finds (one entry per column): Modern/GPT-Finetuned found discrepancy 17, Balanced/GPT found discrepancy 19, and Modern/RNN-Small found discrepancy 24. Fewer certificates are responsible for rare discrepancies (columns with fewer entries have more blue entries), while discrepancies that most models found tend to be triggered by a larger number of certificates (columns with more entries have more green to yellow entries).

Takeaways: Discrepancies and Code Coverage

Several of our MLCerts models find more unique discrepancies in real-world TLS implementations than the baseline state-of-the-art approaches.

At the same time, no single model dominates the other models, suggesting that combining them into ensembles could further increase discovery of discrepancies.

5.2 Certificate Diversity

We now evaluate if we can control the diversity of our synthetic certificates, and whether increased diversity leads to higher code coverage and/or the discovery of more unique discrepancies (RQ2). Our intuition is that diverse TLS certificates (e.g., with unexpected extension combinations) may trigger new discrepancies because validating them requires executing code paths that are otherwise rarely exercised. We can increase the diversity of synthetic certificates by (a) changing the sampling strategy of the models when generating certificates, and (b) generating more synthetic certificates. For practical purposes, we compare these two approaches for the IPv4/RNN-Medium model only, because there was no dominating model and IPv4/RNN-Medium was one of the three best-performing models in the number of unique discrepancies. IPv4/RNN-Medium also has a moderate number of parameters compared to the other two best-performing models. It is more powerful than Modern/RNN-Small, while also being significantly easier to train than GPTs.

5.2.1 Sampling Strategy. When generating certificates, we repeatedly sample from a probability distribution to get output tokens that eventually result in a certificate. For example, if the model outputs a distribution (0.5, 0.4, 0.1) for the three tokens (a, b, c), then the token a is selected as the output token with probability 50%. We can tune the language models with the *temperature* hyperparameter, which controls how uniform the token probability distribution is,

which controls the deviation and diversity of output tokens, and, in turn, of the generated certificates. For instance, for temperature $t = 2.5$, the earlier distribution (0.5, 0.4, 0.1) is scaled to (0.36, 0.34, 0.30) and for $t = 0.1$ it is scaled to (0.72, 0.27, 0.01) instead.

A more uniform sampling strategy will introduce diversity to our synthetic certificates by sampling different tokens, but it will lead to more malformed certificates because more tokens may inadvertently violate the ASN.1 syntax. To understand this trade-off, we evaluate multiple *temperature* values, ranging from 0.5 to 2, and the number of syntactically correct certificates, illustrated in Figure 4. For each value, we attempt to generate 10,000 PEM certificates with our model. We also include the default temperature $t = 1$ (see our detailed results in Section 5.1; the number of unique discrepancies for the default temperature is lower here than in Section 5.1 because we generate 10,000 certificates rather than 100,000). As expected, a lower temperature leads to a more outputs that can be encoded as PEM certificates. Increasing the temperature, and in turn the certificate diversity, helps discovering more discrepancies, up to a certain point, after which most certificates cannot be encoded properly as PEM certificates anymore. Empirically, temperatures $t = 1.25$ and $t = 1.5$ enable us to discover most discrepancies (21), outperforming other temperatures like $t = 0.5$ or $t = 2$, which let us only discover 11 and 9 discrepancies, respectively.

5.2.2 Number of Certificates. We generate up to 100,000 PEM certificates per model for our evaluation of model performance in terms of discovered unique discrepancies (see Section 5.1). An open question is whether 100,000 certificates are sufficient or if the models would discover more unique discrepancies when generating more certificates. To understand whether there is a trend between discrepancies and the number of generated certificates, we investigate the number of required certificates (on average) to trigger a certain number of discrepancies, shown in Figure 5.

We can discover two-thirds of the discrepancies with just 20,000 synthetic certificates. The arc of the curve also indicates diminishing returns in new discrepancies when generating more synthetic certificates. However, there remains a non-zero likelihood of finding more discrepancies with additional certificates and we generate 1,000,000 certificates when investigating security implications next.

Takeaways: Certificate Diversity

The sampling strategy greatly influences certificate diversity and discovered unique discrepancies (up to 2x).

A relatively small number of tens of thousands of MLCerts' synthetic certificates are sufficient to trigger most discrepancies.

5.3 Security Implications

Finally, we investigate the security implications of the discovered discrepancies (RQ3). We first generate a more comprehensive dataset of one million certificates with our IPv4/RNN-Medium model and temperature $t = 1.5$, based our analysis of the best performing models (Section 5.1) and their parameters (Section 5.2). We then analyze the behavioral and security implications of our discrepancies. Particularly noteworthy is that our dataset of 1M certificates generated with IPv4/RNN-Medium covers all 26 discrepancy types we discovered over all models and datasets (Figure 3).

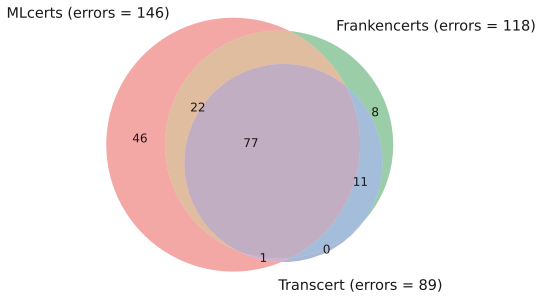


Figure 6: ZLint errors triggered by approach. MLCerts triggers more unique errors (146) than Frankencerts (118) and Transcert (89; combined 119).

We focus on understanding the root causes for the identified discrepancies, to aid discovering bugs or security issues in TLS libraries. Many synthetic certificates may trigger a particular discrepancy, but there might be fewer root causes for them. The underlying root causes are, however, the actual differences between the TLS implementations. Some of these differences can be benign, for example, where the TLS specification allows implementations to differ, but they can also indicate bugs or misunderstandings of the specification, which can have security implications.

We triage and analyze the discovered discrepancies using four techniques, varying in degree of automation, to make large-scale analyses feasible at the necessary technical depth:

5.3.1 Meaningful Information from Output Logs. We extract error messages from the TLS libraries' output logs. The logs for a discrepancy-producing certificate can point to a security flaw if one library rejects the certificate explicitly for a security-relevant reason (e.g., the certificate has expired). This follows because at least one library accepts the certificate (or it would not be a discrepancy). Thus, this is then (a) a mistake by the rejecting libraries, or (b) a security bug by the accepting ones. Specifically, we filter the output logs for security-relevant strings (see our artifact [13, 14]), which we then analyze for their reason of failure:

Certificate Expired or Not Yet Valid: All certificates that were rejected by at least one library due to lifetime validity issues map to a single discrepancy, discrepancy 26 (Figure 3). MatrixSSL accepts the certificates, while all other libraries reject them. Upon closer inspection, we identified a single root cause: MatrixSSL allows for a grace period of 24 hours[1] when comparing a certificate's *notBefore* and *notAfter* times to the local clock. In our experiment, MatrixSSL will also accept certificates that would only be valid on June 16 and June 14 while the other libraries reject the certificates.

Invalid Time Format: We also discovered discrepancies for peculiar time values that some TLS libraries fail to parse successfully. GnuTLS accepts some invalid dates, like February 31st, and both GnuTLS and MatrixSSL accept certificates with leap seconds (value 60 for seconds). We manually analyzed the affected libraries and they reject the certificates if their validity is reasonably far behind (or far ahead) of the local clock (i.e., next day).

[1] See /crypto/keyformat/x509.h#L54-L59 and /crypto/keyformat/x509.c#L5119-L5128.

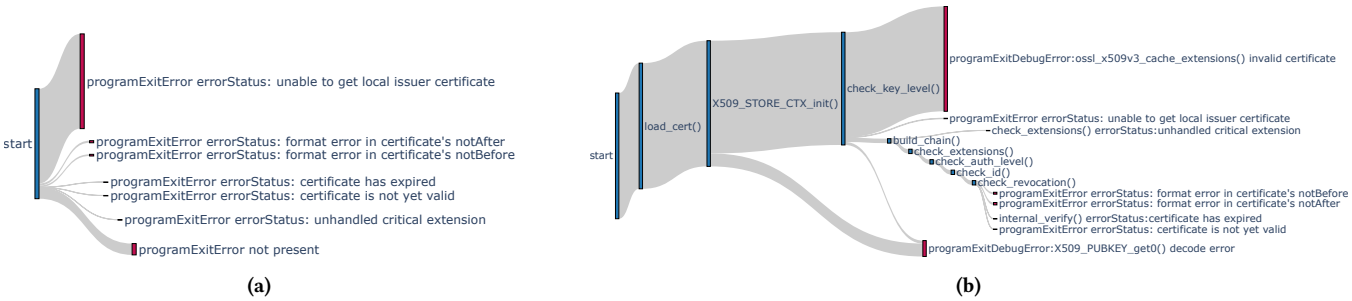


Figure 7: OpenSSL rejections for discrepancy producing certificates with information from (a) output log failure reasons, plus (b) code instrumentation and high level function where an error occured. The vast majority of rejections occur earlier in certificate validation process, before the `build_chain()` function.

Unsupported Critical Extension: Other certificates causing discrepancies include *critical* Certificate Policies extensions with arbitrary data. OpenSSL and GnuTLS accept them while the other libraries reject them. We responsibly disclosed this issue because the specification requires to reject them (RFC5280, Sec. 4.2.1.4 [20]).

This is closely related to, but does not exactly match, a recent security vulnerability in OpenSSL (CVE-2023-0465, moderate severity), which can be abused to make OpenSSL silently skip *all* policy checks. GnuTLS does not offer support for built-in processing of certificate policies and considers this an RFC compliance issue. Similar behavior was also reported by Debnath et al. [35].

Mismatch Between Certificate SKID and IKID: Other discrepancies were caused by certificates with a mismatch between the Subject Key ID (SKID) in the issuer certificate and the Issuer Key ID (IKID) in the leaf certificate. Surprisingly, the TLS specification does not require them to match. Thus, the discrepancy reflects differences in how libraries decided to handle them, and we consider it as benign.

We also find discrepancies for errors related to *Duplicate Extension*, *Unable to Find Issuer*, and *End-entity Certificate Not For TLS Usage* error messages. Our manual analysis did not indicate that they are readily exploitable vulnerabilities.

5.3.2 ZLint. We also leverage ZLint as an oracle to gain insight into the discrepancies and why they are occurring. Figure 6 shows how MLCerts compares to the baselines Frankencerts and Transcert. MLCerts triggers most unique ZLint errors (146), covering ≈ 55% of all ZLint checks. Moreover, MLCerts triggers 46 errors that prior work is unable to trigger, providing evidence that MLCerts can generate diverse TLS certificates to test TLS certificate validation logic. Transcert does not produce a single unique error.

The ZLint checks that get triggered enables us to better understand the implications of our discrepancies. We focus our analysis on the certificates that trigger the ZLint checks that prior work were unable to trigger. A discrepancy already highlights the potential existence of a bug, and by focusing on rare errors that have not been triggered or explored by prior work, we may be able to identify previously unknown issues.

We find 24 ZLint checks from discrepancies and 22 ZLint checks from non-discrepancies (see our artifact [13, 14]). Most are rule violations one could argue are not the responsibility of the library to enforce, but of the issuing CA. For example, the maximum length in a certificate subject's postal code or organization name. However, considering defense in-depth and the many cases of CAs not adhering to the specification, TLS libraries should still want to enforce them. Therefore, we responsibly disclosed our findings to the developers of the TLS libraries (see Section 5.4).

5.3.3 Code Instrumentation. Next, we investigate the root causes of discrepancies by instrumenting the certificate validation logic code. We focus on OpenSSL as an illustrative example, as it is the most widely used TLS library. We add checkpoints at ten different stages of the validation process, to pinpoint the exact stage when OpenSSL rejects a certificate and to obtain more information about the reason for the rejection. We combine this information with the output logs, which may include a reason for the rejection, and we report the function where the error occurred.

We analyze over 25,900 discrepancy-producing certificates that OpenSSL rejects but at least one other library accepts. Figure 7 shows the respective Sankey diagrams for the certificates at different levels of information.

Using only log information containing high-level reasons why OpenSSL rejected the certificate, we see, in Figure 7 (a), that it rejected most certificates for an unknown reason that prevented it from completing a valid chain ("unable to get local issuer certificate"), while it rejected only a small number of certificates with meaningful enough reasons to understand the root cause ("certificate is not yet valid," "unhandled cirtical extension," etc.).

When we include our code instrumentation, shown in Figure 7 (b), we see that OpenSSL rejects the majority of certificates early in the certificate validation process, before `build_chain()` completes. The `build_chain()` function calls lower-level functions that parse various certificate fields. We analyze one such function, `ossl_x509v3_cache_extensions` as a case study. For one of the certificates that OpenSSL rejects with this error, the ASN.1 output from our model for the certificate includes a string of 80 hexadecimal characters (40 bytes) as an extension value. In ASN.1 encoding, extension values can contain arbitrary bytes, and thus it is ASN.1 compliant. However, the extension value is encoded in a way that suggests that it is actually a sequence of 45 bytes. When OpenSSL attempts to parse it, it expects 45 bytes, but only 40 bytes are available, and the function encounters an `ASN1_R_TOO_LONG` error. This shows how MLCerts can produce certificates that violate particular constraints of a valid TLS certificate, which allows us to exercise and test rarely triggered code paths of TLS implementations.

5.3.4 Manual Analysis. Finally, we manually analyze a small set of certificates with discrepancies for which the logs and ZLint do not yield any meaningful information, with goal of providing more details on the root causes for these cases:

Invalid Email in Certificate Subject: OpenSSL accepts certificates that contain badly formatted email addresses in the subject name, like email addresses with multiple @ characters, while the other libraries reject them. Although email addresses are typically not a part of a certificate's subject name, the TLS specification allows for it, for legacy reasons. Interestingly, LibreSSL returns an "unspecified error: should not happen" for these certificates.

v1 Certificate with Extensions: OpenSSL accepts v1 certificates that contain TLS extensions, although extensions are not defined for v1 certificates. Similar behavior was reported by Tian et al. [36].

Parsing Error: LibreSSL accepts certificates with an extra byte in the value of the *KeyUsage* extension, while the other libraries cannot parse them. Debnath et al. reported similar behavior [37].

Takeaways: Security Implications

MLCerts' certificates triggering discrepancies reveal a range of previously unobserved and interesting behavior of TLS libraries.

Certificate validation discrepancies can be benign and the discrepancy information itself is not sufficient to determine the existence of software bugs.

New automated techniques are needed to thoroughly analyze and understand the root causes of discrepancies.

5.4 Disclosures

We discovered implementation bugs in widely deployed TLS libraries that may be security critical. Thus, following best practices, we implemented a coordinated vulnerability disclosure process. For each library, we report the ZLint errors produced by the certificates that the affected library accepts but another library rejects, and we provide the corresponding certificates so that the developers can reproduce our results. All developers except for MatrixSSL, which is no longer maintained, acknowledged our disclosures. The GnuTLS developers are in the process of fixing three issues, prioritized fixing one issue, and reported that one issue was already addressed. LibreSSL and OpenSSL do not consider the issues as their responsibility, but as the responsibility of the CAs to not issue such certificates, contrary to a defense in-depth approach.

5.5 Limitations

Last, we discuss MLCerts' limitations, their impact on our evaluation, and how we attempt to mitigate them.

5.5.1 Training Bias. Language models learn a representation of the X.509 grammar from the training data. Thus, characteristics not in the training data will not be learned (out of distribution), and rarely used features will appear less often. We aim to mitigate this impact by using different training datasets, and improving diversity in synthetic data through our sampling strategy and generating a large number of certificates (see Section 5.2).

5.5.2 Hyperparameters. Our work is fundamentally empirical, that is, our results may not generalize to other TLS libraries and/or different versions or to other implementations of the language models and/or training approaches. To reduce this potential experimentation bias, we evaluate 12 different MLCerts models, using three training datasets and four model architectures (see Section 4.2.1). To encourage further experimentation by other researchers, we also make our code and datasets publicly available [13, 14].

5.5.3 Discrepancies vs. Vulnerabilities. Detecting more discrepancies increases the likelihood of finding vulnerabilities as logic bugs in TLS libraries generally appear as discrepancies. However, not all discrepancies are bugs, and not all bugs are vulnerabilities. Unfortunately, a comprehensive root cause analysis for each discrepancy is complex because a certificate can exhibit multiple issues. While some discrepancies can occur because of features that not all libraries support, these features would be implemented as certificate extensions and they must be marked critical to affect validation. Our analysis indicates that only a negligible amount of certificates are rejected by libraries for such reasons (see Figure 7, third cause from the bottom in the diagram).

Therefore, we only discuss case studies from our manual and semi-automated analyses (see Section 5.3). New approaches for automated root-cause analysis could shed more light on bugs vs. vulnerabilities. Irrespective, our comparison to the state-of-the-art benchmarks Transcert and Frankencerts provides strong empirical evidence for the effectiveness of MLCerts in testing TLS libraries.

5.5.4 Security Impact. While we identified several security issues, our analysis did not reveal a new critical vulnerability. Since TLS libraries are extensively tested, this is expected and matches prior work [11]. Future work on model fine-tuning, for example, using the certificates that cause discrepancies or increase coverage, or through guidance towards untested features, may enable MLCerts to generate certificates focused on vulnerability discovery. Moreover, as TLS/X.509/ASN.1 protocols evolve, MLCerts can continue to test them, because it does not need adapting domain knowledge.

6 Discussion

MLCerts is distinct from Transcert in two major ways: (i) Transcert needs code coverage information at each iteration, making it difficult to parallelize, and (ii) Transcert requires developing certificate mutators based on expert knowledge. MLCerts shows improved performance by not requiring them.

The certificate generation cost for Transcert is reported as 7.91s, but this time increases linearly with each iteration due to coverage graph dependencies. Since each certificate must be processed to bootstrap Transcert, the authors chose to rely on 1,005 seeds to generate 30K certificates. MLCerts (IPv4/RNN-Medium) only takes 1.92s per certificate on a Nvidia Titan X, allowing us to generate 4x as many certificates in the same time. MLCerts has also no internal dependencies and generating certificates is perfectly parallel, enabling us to generate a corpus of 1M certificates quickly.

Transcert missed the discrepancies 4 and 15–19 because of its mutation operators. For discrepancy 4, MLCerts generated two certificates that GnuTLS rejects with a "value is not valid" error in the `gnutls_x509_crt_get_dn` function, because the certificates

have an empty Common Name (CN) in the Subject field. Transcert's mutators do not randomize the certificate attribute values (except for the field notAfter), but they only copy values for the same attribute from other certificates (i.e., perform splicing). Thus, if a value does not already appear in the input set, Transcert cannot generate certificates with such a value at all, which prevents it from discovering discrepancy 4. As for discrepancies 15–19, our inspection shows that OpenSSL throws an "invalid certificate" error in the `ossl_x509v3_cache_extensions` function for all of them, which we discussed as a case study (see Section 5.3.3). Transcert cannot create such certificates because it operates at the X.509 level and the mutation operations can only generate X.509 parse-able certificates, that is, they cannot trigger low-level discrepancies at the ASN.1 level. MLCerts can trigger both types of discrepancies.

Evaluation. We compare MLCerts to Transcert because it is the current state of the art, and to Frankencerts because it is the seminal work. Unfortunately, we cannot compare to other related work (see Section 7): the Mucerts repository is no longer available, NEZHA does not compile [11], and RFCcert and Coveringcert did not release source code. Considering Transcert outperforms RFCcert and NEZHA, and MLCerts outperforms Transcert, we expect that MLCerts will also perform better than them.

We cannot rigorously conclude whether certificate generation using language models is superior (or inferior) to other techniques for testing TLS libraries. However, the empirical evidence that we presented clearly shows the potential of MLCerts and language models to generate certificates that trigger diverse behavior and find novel discrepancies in well-tested software. Ultimately, an ensemble of different certificate generation techniques could provide even more value for improving software security.

Other LLMs and Prompt Engineering. We did not use recent LLM architectures (with billions of parameters) because our results do not indicate that significantly larger models provide any benefit. In fact, we show that classic generative models like RNNs perform as well as GPTs for our use case. In our work, we trained and/or fine-tuned off-the-shelf language models using custom training datasets and built a pipeline for sampling synthetic certificates from the trained representations. Another potential way to generate certificates is to interact directly with a language model such as ChatGPT [38] and provide a prompt relevant to the task of differential testing (e.g., *"Generate a TLS certificate that contains an unexpected combination of extensions"*). While this is an interesting yet orthogonal approach, our preliminary findings show that using such models would require substantial manual effort to write the necessary prompts. We consider this out of scope because we focus on fully automated approaches that scale well.

Using Language Models for Other Input Formats. We use language models for learning representations for X.509 TLS certificates. Given the success of our method in learning representations for software testing, we believe that there is strong potential in using this technique in other domains where software inputs can also be expressed in natural language. Some examples include ASN.1 grammars for data structures used in protocols other than TLS (e.g., 5G), Javascript-based attack vectors, like invalid Content Security Policies, or verbose REST APIs of online services.

7 Related Work

Differential Testing in TLS. Brubaker et al. [7] first introduced Frankencerts, an approach to differentially test TLS libraries. Numerous later work extends on it, aiming to improve certificate generation, such as: Mucerts by using code coverage guidance [8], Coveringcerts by leveraging combinatorial methods [9], SymCerts by incorporating symbolic execution [10], RFCcert by deriving certificate rules from protocol specification documents [36], Transcert by leveraging coverage transfer graphs [11], NEZHA by keeping track of behavioral asymmetries across multiple programs [12], and DRLgencert by using deep reinforcement learning to mutate certificates [39]. In contrast to these synthetic certificate generation techniques, Barenghi et al. [40] rely on a parser with strong termination guarantees using a custom X.509 grammar and Chen et al. [41] focus on finding bugs in certificate parsing modules of the libraries. Transcert performs best and is state of the art, as it outperforms Frankencerts by 18x, RFCcerts by 10x and NEZHA by 1.43x. Our approach, MLCerts, outperforms Transcert.

Language Models for Software Security. Leveraging language models to improve software security is an active area of research. Godefroid et al. [22] use generative language models, before the advent of LLMs, to generate fuzzing inputs. Sablotny et al. [42] use RNNs to generate HTML to fuzz browsers, but they did not use differential testing and did not find any bugs. Other approaches use LLMs for test case generation and mutation, like ChatAFL [43], CodaMosa [44], TitanFuzz [45], FuzzGPT [46] or ChatFuzz [47]. Unlike our work, they do not train a small model from scratch, but they use off-the-shelf LLMs. Jiang et al. [48] investigate how LLMs are used for fuzzing and provide guidance for prompt design.

8 Conclusion

In this paper, we introduced MLCerts, an approach using generative language models for learning X.509 TLS certificate representations. Our results show the potential of MLCerts to generate synthetic certificates that trigger diverse behavior during differential testing and can help discover previously unknown issues in TLS libraries. We also evaluated how to increase the diversity of the generated certificates, assessed the certificate corpus performance on different metrics, and analyzed the discrepancies to understand their root causes. Fundamentally, our work motivates the use of language models for differential testing in other domains, and how one can use machine learning to not only learn input grammars, but to also learn the peculiar ways in which deviations from the input grammar and "hallucinations" can enable better software testing.

Acknowledgments

This work is based on research supported by the National Science Foundation (NSF Grant 1955227), the Internet Society Foundation, the Vienna Science and Technology Fund (WWTF) and the City of Vienna [Grant ID: 10.47379/ICT19056 and 10.47379/ICT22060], the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, the ICANN Grant Program, and SBA Research (SBA-K1 NGC). Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] International Telecommunication Union (ITU). *X.509: Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks*. URL: https://www.itu.int/rec/T-REC-X.509/en (visited on 11/14/2024).

[2] OSSFuzz, M. Caswell, and R. Levitte. *CVE-2023-2650: Possible DoS translating ASN.1 object identifiers*. May 30, 2023. URL: https://www.cve.org/CVERecord?id=CVE-2023-2650 (visited on 09/07/2024).

[3] Polar Bear and P. Dale. *CVE-2022-3602: X.509 Email Address 4-byte Buffer Overflow*. Nov. 1, 2022. URL: https://www.cve.org/CVERecord?id=CVE-2022-3602 (visited on 11/25/2025).

[4] N. Mehta. *CVE-2014-0160: A missing bounds check in the handling of the TLS heartbeat extension can be used to reveal up to 64kB of memory to a connected client or server (a.k.a. Heartbleed)*. Apr. 7, 2014. URL: https://www.cve.org/CVERecord?id=CVE-2014-0160 (visited on 11/25/2025).

[5] OSSFuzz, OpenAI Security Research Team, and A. Hamilton. *CVE-2025-32989: Gnutls: vulnerability in gnutls sct extension parsing*. July 10, 2025. URL: https://www.cve.org/CVERecord?id=CVE-2025-32989 (visited on 11/25/2025).

[6] A. Langley (ImperialViolet). *Apple's SSL/TLS bug*. Feb. 22, 2014. URL: https://www.imperialviolet.org/2014/02/22/applebug.html (visited on 11/25/2025).

[7] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. "Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations." In: *Proceedings of the 35th IEEE Symposium on Security & Privacy (S&P)*. May 2014. DOI: 10.1109/SP.2014.15.

[8] Y. Chen and Z. Su. "Guided Differential Testing of Certificate Validation in SSL/TLS Implementations." In: *Proceedings of the 10th ACM International Conference on the Foundations of Software Engineering (FSE)*. Aug. 2015. DOI: 10.1145/2786805.2786835.

[9] K. Kleine and D. E. Simos. "Coveringcerts: Combinatorial Methods for X.509 Certificate Testing." In: *Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Mar. 2017. DOI: 10.1109/ICST.2017.14.

[10] S. Y. Chau, O. Chowdhury, E. Hoque, H. Ge, A. Kate, C. Nita-Rotaru, and N. Li. "SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations." In: *Proceedings of the 38th IEEE Symposium on Security & Privacy (S&P)*. May 2017. DOI: 10.1109/SP.2017.40.

[11] P. Nie, C. Wan, J. Zhu, Z. Lin, Y. Chen, and Z. Su. "Coverage-directed Differential Testing of X. 509 Certificate Validation in SSL/TLS Implementations." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 32.1 (Jan. 2023). DOI: 10.1145/3510416.

[12] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. "NEZHA: Efficient Domain-Independent Differential Testing." In: *Proceedings of the 38th IEEE Symposium on Security & Privacy (S&P)*. May 2017. DOI: 10.1109/SP.2017.27.

[13] M. T. Paracha, K. Posluns, K. Borgolte, M. Lindorfer, and D. Choffnes, *MLCerts – Source Code*. URL: https://github.com/rub-softsec/MLCerts.

[14] M. T. Paracha, K. Posluns, K. Borgolte, M. Lindorfer, and D. Choffnes. *MLCerts – Datasets and Language Models*. DOI: 10.5281/zenodo.15971208.

[15] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Aug. 2018. DOI: 10.17487/RFC8446.

[16] E. Rescorla and T. Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. Aug. 2008. DOI: 10.17487/RFC5246.

[17] International Telecommunication Union (ITU). *X.680: Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation*. URL: https://www.itu.int/rec/T-REC-X.680 (visited on 11/25/2025).

[18] International Telecommunication Union (ITU). *X.690: Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*. URL: https://www.itu.int/rec/T-REC-X.690 (visited on 11/25/2025).

[19] S. Josefsson and S. Leonard. *Textual Encodings of PKIX, PKCS, and CMS Structures*. Apr. 2015. DOI: 10.17487/RFC7468.

[20] S. Boeyen, S. Santesson, T. Polk, R. Housley, S. Farrell, and D. Cooper. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. May 2008. DOI: 10.17487/RFC5280.

[21] B. Swierzy, F. Boes, T. Pohl, C. Bungartz, and M. Meier. "SoK: Automated Software Testing for TLS Libraries." In: *Proceedings of the 19th International Conference on Availability, Reliability and Security (ARES)*. July 2024. DOI: 10.1145/3664476.3670871.

[22] P. Godefroid, H. Peleg, and R. Singh. "Learn&Fuzz: Machine learning for Input Fuzzing." In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Oct. 2017. DOI: 10.1109/ASE.2017.8115618.

[23] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. "Attention is All you Need." In: *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS)*. Dec. 2017. DOI: 10.5555/3295222.3295349.

[24] S. Hochreiter and J. Schmidhuber. "Long Short-Term Memory." In: *Neural Computation* 9.8 (Nov. 1997). DOI: 10.1162/neco.1997.9.8.1735.

[25] J. Schmidhuber. *The 2010s: Our Decade of Deep Learning / Outlook on the 2020s*. Feb. 20, 2020. URL: https://people.idsia.ch/~juergen/2010s-our-decade-of-deep-learning.html (visited on 11/25/2025).

[26] Z. Durumeric, E. Wustrow, and J. A. Halderman. "ZMap: Fast Internet-wide Scanning and Its Security Applications." In: *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*. Aug. 2013. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/durumeric.

[27] Project Sonar | Rapid7. *SSL Certificates: X.509 certificate metadata observed when communicating with HTTPS endpoints*. URL: https://opendata.rapid7.com/sonar.ssl/ (visited on 07/09/2024).

[28] p1-bmu and contributor, *pycrate – A Python library providing a runtime for encoding and decoding data structures, including CSN.1 and ASN.1*. Version 0.5.5, June 2022. URL: https://github.com/pycrate-org/pycrate.

[29] N. Barhate (nikhilbarhate99), *Char RNN PyTorch – Minimalist code for character-level language modelling using Multi-layer Recurrent Neural Networks (LSTM) in PyTorch*. Version 2cf836f, July 2019. URL: https://github.com/nikhilbarhate99/Char-RNN-PyTorch.

[30] S. Black, G. Leo, P. Wang, C. Leahy, and S. Biderman, *GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow* version 1.0, Mar. 2021. DOI: 10.5281/zenodo.5297715.

[31] GNU Compiler Collection, *GCOV coverage testing tool* version 11.4.0, May 2023. URL: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[32] W. Hommel (wolfcw) and contributors, *libfaketime* version 0.9, Feb. 2011. URL: https://github.com/wolfcw/libfaketime.

[33] C. Henderson (christopher-henderson) and contributors, *ZLint – A X.509 Certificate Linter focused on Web PKI Standards and Requirements* version 3.0.0-rc1, Nov. 2020. URL: https://github.com/zmap/zlint.

[34] CAB Forum. *Certification Authorities, Web Browsers, and Interested Parties Working to Secure the Web*. URL: https://cabforum.org/ (visited on 11/25/2025).

[35] J. Debnath, C. Jenkins, Y. Sun, S. Y. Chau, and O. Chowdhury. "ARMOR: A Formally Verified Implementation of X. 509 Certificate Chain

Validation." In: *Proceedings of the 45th IEEE Symposium on Security & Privacy (S&P)*. May 2024. DOI: 10.1109/SP54263.2024.00220.

[36] C. Tian, C. Chen, Z. Duan, and L. Zhao. "Differential Testing of Certificate Validation in SSL/TLS Implementations: An RFC-guided Approach." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28.4 (Oct. 2019). DOI: 10.1145/3355048.

[37] J. Debnath, S. Y. Chau, and O. Chowdhury. "On Re-engineering the X. 509 PKI with Executable Dpecification for Better Implementation Guarantees." In: *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Nov. 2021. DOI: 10.1145/3460120.3484793.

[38] OpenAI. *ChatGPT*. URL: https://chatgpt.com/ (visited on 11/25/2025).

[39] C. Chen, W. Diao, Y. Zeng, S. Guo, and C. Hu. "DRLgencert: Deep Learning-Based Automated Testing of Certificate Verification in SSL/TLS Implementations." In: *Proceedings of the34th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Sept. 2018. DOI: 10.1109/ICSME.2018.00014.

[40] A. Barenghi, N. Mainardi, and G. Pelosi. "Systematic Parsing of X.509: Eradicating Security Issues with a Parse Tree." In: *Journal of Computer Security* 26.6 (Apr. 2018). DOI: 10.3233/JCS-171110.

[41] C. Chen, P. Ren, Z. Duan, C. Tian, X. Lu, and B. Yu. "SBDT: Search-Based Differential Testing of Certificate Parsers in SSL/TLS Implementations." In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. July 2023. DOI: 10.1145/3597926.3598110.

[42] M. Sablotny, B. S. Jensen, and C. W. Johnson. "Recurrent Neural Networks for Fuzz Testing Web Browsers." In: *Proceedings of the 28th Annual Conference on Information Security and Cryptology (ICISC)*. Nov. 2018. DOI: 10.1007/978-3-030-12146-4_22.

[43] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury. "Large Language Model guided Protocol Fuzzing." In: *Proceedings of the 31st Network and Distributed System Security Symposium (NDSS)*. Feb. 2024. DOI: 10.14722/ndss.2024.24556.

[44] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen. "Codamosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models." In: *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. May 2023. DOI: 10.1109/ICSE48619.2023.0008.

[45] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang. "Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models." In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. July 2023. DOI: 10.1145/3597926.3598067.

[46] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang. "Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries." In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*. Apr. 2024. DOI: 10.1145/3597503.3623343.

[47] M. Rostami, M. Chilese, S. Zeitouni, R. Kande, J. Rajendran, and A.-R. Sadeghi. "Beyond Random Inputs: A Novel ML-Based Hardware Fuzzing." In: *Proceedings of the 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Mar. 2024. DOI: 10.23919/DATE58400.2024.10546625.

[48] Y. Jiang, J. Liang, F. Ma, Y. Chen, C. Zhou, Y. Shen, Z. Wu, J. Fu, M. Wang, S. Li, and Q. Zhang. "When Fuzzing Meets LLMs: Challenges and Opportunities." In: *Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE)*. July 2024. DOI: 10.1145/3663529.3663784.