

# On Evaluating Fuzzers with Context-Sensitive Fuzzed Inputs: A Case Study on PKCS#1-v1.5

S Mahmudul Hasan, Polina Kozyreva, and Endadul Hoque  
Syracuse University, USA

**Abstract**—Cryptographic standards like the PKCS#1-v1.5 signature scheme for RSA are essential for secure digital communications, yet cryptographic libraries remain vulnerable. Fuzzing, a security testing technique, often struggles to detect memory-safety bugs in these libraries due to the need for context-sensitive inputs—those with complex semantic relationships between their fields. This paper presents a preliminary study that evaluates 7 fuzzers for their ability to generate such inputs across 5 libraries implementing the PKCS#1-v1.5 signature verification scheme. Our evaluation highlights the limitations of fuzzers with context-sensitive inputs. Black-box fuzzers supporting semantic constraints like ISLA (100% valid inputs) and Morpheus (57%) outperform grey-box fuzzers such as Nautilus (37%), AFL++ (31%) and AFL (29%), which struggle with code coverage focus.

**Index Terms**—Fuzzing, Cryptographic protocols, PKCS#1-v1.5

## I. INTRODUCTION

Cryptographic protocols are essential to ensuring security, yet many libraries implementing these protocols remain vulnerable [1–4]. Fuzzing—a widely used security testing technique—has uncovered numerous flaws [5]. However, robust, general-purpose fuzzers like AFL and its variants [5, 6] still struggle to detect vulnerabilities in cryptographic libraries [1–3]. This limitation applies notably to the PKCS#1-v1.5 signature verification scheme [7], a fundamental cryptographic primitive used to verify RSA digital signatures in protocols like TLS and IPsec, which secure digital communications.

Recent research [1, 2] has shown that customized tools specifically designed to find signature forgery vulnerabilities in PKCS#1-v1.5 libraries unexpectedly uncovered numerous memory-safety bugs (e.g., buffer overflows) that were missed by AFL [8], despite its strong track record in detecting such issues. This finding motivated us to ask a broader question: Whether other general-purpose fuzzers—varying in approach from black-box to grey-box and using techniques like input grammars or seed inputs—could perform any better in identifying these bugs.

Further inspection reveals that the major challenge in triggering these bugs lies in crafting test inputs that not only contain specific “jackpot” values but also pass the strict input validations typical of cryptographic protocols [7, 9]. These validations involve complex sanity checks at early stages in the code, allowing only correctly formatted inputs to proceed to deeper layers. Protocol standards dictate these semantic input validation requirements, which are often *context-sensitive*, reflecting intricate relationships between various fields within the input. Consequently, an incorrectly formatted input that fails

to meet these semantic requirements will likely be discarded before it reaches the buggy code.

This indicates that a fuzzer capable of generating semantically well-formatted inputs would be more effective in uncovering such bugs. To evaluate the suitability of state-of-the-art general-purpose fuzzers for testing PKCS#1-v1.5 libraries, our study focuses on assessing their ability to produce *context-sensitive fuzzed inputs*—those that are semantically well-formatted but include values mutated by the fuzzer.

In this paper, we present a preliminary study evaluating the ability of 7 modern fuzzers to generate context-sensitive fuzzed inputs for 5 libraries implementing the PKCS#1-v1.5 signature verification scheme. These fuzzers employ a range of approaches, from grey-box and black-box methods to techniques like seed inputs and input grammars. While previous research has used coverage-based metrics, such as branch coverage, to assess fuzzers [5, 10–13], we use the validity rate of context-sensitive fuzzed inputs as our primary metric. Unlike branch coverage, which relies on source code access or instrumentation and is thus unsuitable for black-box fuzzers and closed-source libraries, the validity rate offers a source-independent metric for assessing a fuzzer’s ability to meet PKCS#1-v1.5 semantic requirements.

Our evaluation shows that black-box fuzzers supporting semantic constraints (e.g., Morpheus [2], ISLA [14]) outperform grey-box fuzzers (e.g., AFL++ [15], Nautilus [16]), which often get sidetracked by their focus on code coverage. However, AFL in deterministic mode can surpass context-free grammar-based fuzzers (e.g., Peach [17]) by selectively mutating non-essential parts of context-sensitive inputs. Similar challenges in handling PKCS#1-v1.5 inputs likely apply to protocols like TCP and DNS, highlighting the need for further research. Our tool is available at <https://github.com/syne-lab/fuzz-eval>.

## II. PRELIMINARIES

This section briefly covers fuzzing basics and PKCS#1-v1.5 standard for RSA signature generation and verification.

**Fuzzing.** Fuzzing is a powerful software testing method that feeds abnormal inputs to programs, exposing crashes and vulnerabilities [6]. Fuzzers come in three main types: white-box, grey-box, and black-box. White-box fuzzers (e.g., SAGE [18]) use detailed program knowledge through symbolic execution to analyze path constraints. Grey-box fuzzers, like AFL and AFL++, rely on partial insights such as code coverage metrics. In contrast, black-box fuzzers (e.g., Peach, Morpheus) lack

internal program knowledge. Fuzzers use different input generation methods, from random mutations [8, 15] to grammar-based techniques [14, 16, 17, 19] that leverage input structure and relationships for more targeted testing.

**PKCS#1-v1.5 for RSA Signatures.** PKCS#1-v1.5 [7], a cryptographic standard, plays a pivotal role in secure communication protocols such as SSL/TLS and IPsec as well as in software signing and X.509 certificates. It defines formats for RSA encryption and signature schemes, including padding schemes. To generate a signature for a message,  $M$ , using RSA algorithm and PKCS#1-v1.5 padding scheme, the message  $M$  is first encoded to produce an EM structure as follows:

$$EM = 0x00 \parallel BT \parallel PS \parallel 0x00 \parallel PL$$

Here, BT represents the block type (0x01 for signatures, 0x02 for encryption), PS denotes a padding string (byte sequence containing 0xFF) for RSA signatures, and PL represents the payload bytes, *i.e.*, a hash digest of  $M$ . The signature,  $S$ , is generated using the operation  $S = EM^d \bmod n$ , where  $d$  is the RSA private exponent and  $n$  is the public modulus.

To ensure the generation of valid EM structures, it is crucial to maintain  $|EM| = |n|$ , where  $|n|$  denotes the size of the public modulus in bytes. Additional constraints on the EM structure include the minimum length of the padding string ( $|PS| \geq 8$ ) and the relationship between PS and PL ( $|PS| = |n| - |PL| - 3$ ). In our study, we assess fuzzers' effectiveness in generating various EM structures that satisfy these PKCS#1-v1.5 constraints.

### III. DESIGN AND IMPLEMENTATION

#### A. Design

We developed CSFuzz, a unified platform to evaluate fuzzers' ability to generate context-sensitive inputs. CSFuzz standardizes testing conditions across different fuzzing campaigns through customizable settings and consists of three main components: the *Controller*, *Validator*, and *Oracle*. The controller manages fuzzing campaigns, while the validator receives the fuzzer-generated inputs to verify them using the oracle based on the PKCS#1-v1.5 standard and records results.

*a) Controller:* The Controller uses a configuration file to set up campaign details, such as the fuzzer and library, fuzzing duration and seed. After parsing the configuration, it launches the validator server, starts the (fuzzer, library) campaign, monitors progress, and shuts down the campaign upon completion. The controller also supports concurrent campaigns.

*b) Validator:* The validator, a locally hosted TCP server, receives fuzzer-generated inputs from test harnesses. Its main role is to validate these inputs by consulting the oracle. For each input, the validator queries the oracle and logs the response.

*c) Oracle:* The oracle checks if a fuzzer-generated input satisfies PKCS#1-v1.5 constraints. Although the payload (PL) contains a cryptographic hash of the message, the oracle treats it simply as a byte sequence without checking the validity of the hash digest. Our focus here is to confirm that each generated input (EM) satisfies these constraints: (a)  $|EM| = |n|$ , (b)  $|PS| \geq 8$ , and (c)  $|PS| = |n| - |PL| - 3$  (see § II).

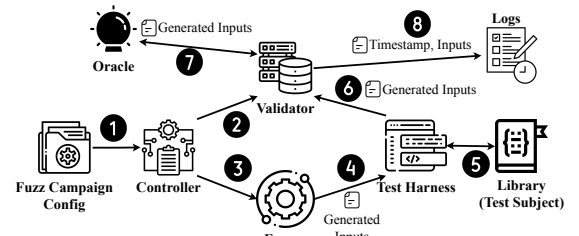


Fig. 1. Workflow of CSFuzz

**Fuzzer, Test Harness, and Test Subject.** Instead of directly fuzzing the test subject (*i.e.*, a PKCS#1-v1.5 library), we configure the fuzzer to target a test harness that uses the library for RSA signature verification. The fuzzer supplies the harness with a byte array (*i.e.*, an EM structure) to generate an RSA signature. Inspired by [1, 2], key RSA components (public modulus, public exponent, and private exponent) are kept constant across all tests. After signature generation, the harness invokes the library's signature verification function. Once the function returns, the harness forwards the EM structure to CSFuzz's validator to check PKCS#1-v1.5 compliance. This process repeats with each new EM from the fuzzer.

**Workflow.** Fig. 1 depicts CSFuzz's operation. In summary, the controller takes in the fuzz campaign configuration (❶), launches the validator server (❷), initiates the (fuzzer, library) fuzzing campaign (❸), where the fuzzer provides inputs to the test harness (❹). The test harness processes each input, invokes the signature verification function (❺), and sends the input to the validator (❻). The validator then consults the oracle to validate the input (❼) and logs the response (❽).

#### B. Implementation

CSFuzz is primarily implemented in Python3. The controller, written in Python3, processes a TOML configuration file. For TCP functionality, the validator and controller utilize the Twisted library in Python3. The critical oracle component is a Python3 function. However, we developed test harnesses for each PKCS#1-v1.5 library in C/C++. For reproducibility, we employed a Docker-container-based virtualized environment, dedicating each container to a single fuzzing campaign.

### IV. EVALUATION

We evaluate the effectiveness of fuzzers in generating context-sensitive mutated inputs for PKCS#1-v1.5 signature verification libraries. Inputs satisfying PKCS#1-v1.5 semantic constraints are deemed *valid* by our oracle. Effectiveness is measured as the percentage of valid inputs each fuzzer produces over time—higher is better. Metrics like code coverage and bug detection are left for future work, but fuzzer throughput (valid/invalid inputs per second) is reported in [20].

#### A. Experimental Setup

**Fuzzers.** Our objective was to evaluate a wide range of fuzzers, from black-box to grey-box approaches, utilizing various techniques such as input grammars or seed inputs, and differing in their need for access to source code. We assessed

7 fuzzers: AFL [8], AFL++ [15], AFLSmart [19], Nautilus [16], Peach [17], Morpheus [2], and ISLA [14].

AFL, AFL++, AFLSmart, and Nautilus are grey-box fuzzers. While AFL, AFL++, and AFLSmart benefit from valid seed inputs, Nautilus requires the input grammar.<sup>1</sup> Peach and ISLA require input format specifications, with Peach operating as a black-box fuzzer and ISLA functioning as an input generator for any black-box fuzzer. Morpheus, another black-box fuzzer, is specifically designed for testing PKCS#1-v1.5 implementations, embedding these specifications internally.

**PKCS#1-v1.5 Libraries.** We tested the fuzzers on 5 widely used libraries implementing PKCS#1-v1.5 signature verification, all developed in C/C++ and downloaded from their respective public repository. Instead of fuzzing each library directly, we fuzzed a test harness that uses the library for RSA signature verification. The harness takes a PKCS#1-v1.5 formatted byte array (an EM structure, see § II), generates an RSA signature using the EM, a predefined private key ( $n, d$ ), and a message (*e.g.*, "hello world!"), and then verifies the RSA signature via the library. The harness then sends the original EM structure to CSFuzz’s validator to check its validity, not the signature or the library’s cryptographic output.

**Computing Setup.** Each fuzzer was tested for 10 hours on each library across five repetitions, totaling about 1800 computing hours. Tests were conducted on a third-party cloud server, with each campaign running in a dedicated Ubuntu 20.04 Docker container with 8GB RAM and 3 cores. To keep costs under \$500 USD, we limited testing to 10 hours, based on cloud provider estimates for computation and storage.

## B. Results

Table I presents the percentage of valid inputs generated by each fuzzer for each library. Each percentage is the average from five repetitions of each fuzzing campaign. Note that ISLA and Morpheus results are not included in this table.

ISLA is distinctive in consistently generating 100% valid inputs. This is achieved by using the EM structure’s context-sensitive grammar along with an SMT solver [21] to resolve EM’s constraints during input generation, resulting in a flawless output of valid inputs across all PKCS#1-v1.5 libraries.

Morpheus follows ISLA as the next most effective fuzzer, producing 56.69% valid inputs for all libraries. Operating based on PKCS#1-v1.5 specifications, Morpheus generates a fixed set of inputs through predefined mutations, regardless of the library under test. Consequently, it yields fewer valid inputs than ISLA.

Peach, a black-box fuzzer, proves to be the least effective, generating only 17.81% valid inputs on average. Although it relies on input specifications, Peach only supports a context-free grammar of the EM structure, limiting its performance since it lacks awareness of EM’s necessary constraints.

Among grey-box fuzzers, Nautilus generates the most valid inputs overall (36.71%), while AFL and AFL++ produce

<sup>1</sup>We used AFL (v2.57b) and AFL++ (v4.07c) from their GitHub repositories. We ran AFL++ using its default configuration without enabling advanced features like custom mutators or grammar support.

TABLE I  
THE PERCENTAGE OF VALID INPUTS GENERATED BY THE FUZZERS  
(EXCLUDING MORPHEUS AND ISLA) ACROSS PKCS#1-v1.5 LIBRARIES.  
A HIGHER VALUE INDICATES GREATER EFFECTIVENESS.

Libraries	AFL	AFL++	AFLSmart	Nautilus	Peach
axtls	24.58	25.70	0.58	18.94	17.54
hostapd	27.36	33.41	3.69	42.58	17.86
libtomcrypt	34.07	38.43	6.88	41.69	17.87
matrixssl	28.88	25.13	0.84	26.66	17.90
wpasupplicant	29.18	31.59	3.62	53.70	17.89
<b>Average</b>	28.81	30.85	3.12	36.71	17.81

28.81% and 30.85% valid inputs, respectively. Nautilus’s higher performance is attributed to its additional requirement for a context-free grammar of EM, giving it an advantage over the other two fuzzers.

AFLSmart, despite requiring input specifications, is less effective than other AFL-based fuzzers. For instance, while Nautilus generated 18.94% valid inputs for axtls, AFLSmart produced only 0.58% for the same target. This disparity is due to AFLSmart’s heavy reliance on havoc-based mutations, which significantly alter even valid seed inputs.

Overall, we observed that black-box fuzzers or input generators that rely on input specifications with semantic constraints, such as Morpheus and ISLA, generally outperform grey-box fuzzers. In contrast, grey-box fuzzers tend to be derailed by their focus on increasing code coverage, even when they start with strong seed inputs (AFL++), context-free input grammars (Nautilus), or both (AFLSmart).

## V. TAKEAWAYS

**Customized vs. General-purpose Tools.** Although customized tools like Morpheus are generally expected to excel, general-purpose tools like ISLA can outperform Morpheus due to their support for context-sensitive constraints and use of an SMT solver. However, ISLA’s advantage comes with a trade-off: slower input generation.

**Context-free Input Grammars.** Fuzzers relying on context-free input specifications, like Nautilus, AFLSmart, and Peach, struggle to satisfy context-sensitive constraints, leading to numerous invalid inputs.

**Deterministic Fuzzing Mode.** AFL-like grey-box fuzzers in deterministic mode, given valid seed inputs, can outperform grammar-based fuzzers like Peach by selectively mutating non-essential parts of the EM structure (*e.g.*, the payload (PL), which is irrelevant to our oracle). This approach helps them discover new paths while preserving valid inputs. In contrast, disabling deterministic mode, as in AFLSmart, often disrupts the input structure and increases invalid inputs.

**Influence of Test-subjects’ Code.** Coverage-guided grey-box fuzzers, regardless of reliance on input specifications, are influenced by each test subject’s code. The rate of valid input generation varies with both the test subject and time.

**Threat to Validity.** Testing 7 fuzzers across 5 libraries for 10 hours may limit insights into their full potential for generating valid inputs, affecting generalizability.

## VI. RELATED WORK

**Fuzzer Evaluation.** LAVA [22] introduces synthetic bugs for testing, though these may differ from real-world vulnerabilities [10, 11]. Evil Coder [23] uses data flow analysis to insert bugs, while other benchmarks like CGC binaries [24], Magma [25], UniFuzz [26] and Fuzzbench [27] incorporate real-world vulnerabilities. Similarly, our study assesses fuzzers on cryptographic libraries implementing the PKCS#1-v1.5 signature scheme, aligning with real-world applications. Although our selected 5 libraries may not form a comprehensive benchmark, we acknowledge the ongoing challenge of creating one.

Unlike prior studies [5, 10–13] that prioritize metrics like code or branch coverage and known-bug counts, our work uniquely compares fuzzers based on their effectiveness in generating inputs that meet complex semantic constraints. Traditional coverage and bug metrics can be unreliable [28] as they may overlook a fuzzer’s depth in exploring program states. For instance, a fuzzer for XML might find parsing bugs but miss core logic issues. By focusing on semantically valid inputs, our study emphasizes a fuzzer’s capacity to reach deeper program states.

**Evaluation of PKCS#1-v1.5.** Automated tools have uncovered signature forgery vulnerabilities [29] in several libraries using symbolic execution [1] and domain-specific fuzzers [2]. In contrast, our study focuses on inputs that trigger memory-safety bugs protected by initial validation [2, 4], which must satisfy context-sensitive semantic constraints. We examined fuzzers’ ability to generate such inputs for PKCS#1-v1.5.

## VII. CONCLUSION AND FUTURE WORK

Our study of 7 popular fuzzers reveals their effectiveness in generating context-sensitive inputs for testing PKCS#1-v1.5. The findings highlight performance differences and the challenges fuzzers face with context-sensitive inputs. In the future, we will conduct a large-scale evaluation to assess fuzzers on additional performance metrics, such as code coverage and bug detection, while generating context-sensitive inputs.

## ACKNOWLEDGMENT

This research was supported, in part, by the NSF award CNS-2339350 and a gift from Google Research.

## REFERENCES

- [1] S. Y. Chau, M. Yahyazadeh, O. Chowdhury, A. Kate, and N. Li, “Analyzing Semantic Correctness with Symbolic Execution: A Case Study on PKCS#1 v1.5 Signature Verification,” in *NDSS*, 2019.
- [2] M. Yahyazadeh, S. Y. Chau, L. Li, M. H. Hue, J. Debnath, S. C. Ip, C. N. Li, E. Hoque, and O. Chowdhury, “Morpheus: Bringing the (pkcs) one to meet the oracle,” in *Proc. of ACM CCS*, 2021.
- [3] S. 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 *Proc. of IEEE S&P*, 2017.
- [4] 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 *Proc. of IEEE S&P*, 2014.
- [5] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, “The art, science, and engineering of fuzzing: A survey,” *IEEE TSE*, vol. 47, no. 11, 2019.
- [6] M. Boehme, C. Cadar, and A. Roychoudhury, “Fuzzing: Challenges and reflections,” *IEEE Software*, vol. 38, no. 3, 2021.
- [7] B. Kaliski, “PKCS #1: RSA Encryption Version 1.5,” RFC 2313, 1998, <https://www.rfc-editor.org/info/rfc2313>.
- [8] M. Zalewski, “American fuzzy lop,” <https://lcamtuf.coredump.cx/afl/>, 2020.
- [9] T. Dierks and E. Rescorla, “The transport layer security (tls) protocol version 1.2,” Internet Requests for Comments, RFC 5246, 2008, <http://www.rfc-editor.org/rfc/rfc5246.txt>.
- [10] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating fuzz testing,” in *Proc. of ACM CCS*, 2018.
- [11] J. Bundt, A. Fasano, B. Dolan-Gavitt, W. Robertson, and T. Leek, “Evaluating synthetic bugs,” in *ACM ASIACCS*, 2021.
- [12] P. Görz, B. Mathis, K. Hassler, E. Güler, T. Holz, A. Zeller, and R. Gopinath, “Systematic assessment of fuzzers using mutation analysis,” in *Proc. of USENIX Security*, 2023.
- [13] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, “Sok: Prudent evaluation practices for fuzzing,” in *IEEE S&P*, 2024.
- [14] D. Steinhöfel and A. Zeller, “Input invariants,” in *Proc. of ACM ESEC/FSE*, 2022.
- [15] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, “AFL++: Combining incremental steps of fuzzing research,” in *Proc. of USENIX WOOT*, 2020.
- [16] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, “Nautilus: Fishing for deep bugs with grammars,” in *NDSS*, 2019.
- [17] GitLab, “Peach fuzzer 3,” <https://peachtech.gitlab.io/peach-fuzzer-community/>, 2022.
- [18] P. Godefroid, M. Y. Levin, and D. Molnar, “Sage: whitebox fuzzing for security testing,” *CACM*, vol. 55, no. 3, 2012.
- [19] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, “Smart greybox fuzzing,” *IEEE TSE*, vol. 47, no. 9, 2019.
- [20] S. Hasan, P. Kozyreva, and E. Hoque, “Fuzzeval: Assessing fuzzers on generating context-sensitive inputs,” *arXiv preprint arXiv:2409.12331*, 2024.
- [21] “The z3 theorem prover,” <https://github.com/Z3Prover/z3>.
- [22] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, “Lava: Large-scale automated vulnerability addition,” in *Proc. of IEEE S&P*, 2016.
- [23] J. Pewny and T. Holz, “Evilcoder: automated bug insertion,” in *Proc. of ACSAC*, 2016.
- [24] “Darpa cyber grand challenge (cgc) binaries,” <https://github.com/CyberGrandChallenge/>.
- [25] A. Hazimeh, A. Herrera, and M. Payer, “Magma: A ground-truth fuzzing benchmark,” *ACM POMACS*, vol. 4, no. 3, 2020.
- [26] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng *et al.*, “UNIFUZZ: A holistic and pragmatic Metrics-Driven platform for evaluating fuzzers,” in *Proc. of USENIX Security*, 2021.
- [27] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, “Fuzzbench: an open fuzzer benchmarking platform and service,” in *Proc. of ACM ESEC/FSE*, 2021.
- [28] M. Böhme, L. Szekeres, and J. Metzman, “On the reliability of coverage-based fuzzer benchmarking,” in *Proc. of ICSE*, 2022.
- [29] H. Finney, “Bleichenbacher’s rsa signature forgery based on implementation error,” <https://mailarchive.ietf.org/arch/msg/openpgp/5rnE9ZRN1AokBVj3VqblGIP63QE/>, 2006.