How Can ChatGPT Support Human Security Testers to Help Mitigate Supply Chain Attacks?

Ying Zhang, Wenjia Song, Zhengjie Ji, Danfeng (Daphne) Yao, Na Meng

***Abstract*—Developers often build software on top of third-party libraries (Libs) to improve programmer productivity and software quality. The libraries may contain vulnerabilities exploitable by hackers to attack the applications (Apps) built on top of them. Such attacks are known as software supply chain attacks, the documented number of which has increased 742% in 2022. Researchers and developers created tools to mitigate such attacks, by scanning the library dependencies of Apps, identifying the usage of vulnerable library versions, and suggesting secure alternatives to vulnerable dependencies. However, recent studies show that many developers do not trust the reports by these tools; they need code or evidence to demonstrate how library vulnerabilities lead to security exploits, in order to assess vulnerability severity and modification necessity. Unfortunately, manually crafting demos of application-specific attacks is challenging and time-consuming, and there is insufficient tool support to automate that procedure.**

To help developers enhance software security, in this study, we systematically explored the usage of a large language model (LLM)—ChatGPT-4.0—to generate security tests, which unit tests demonstrate how vulnerable library dependencies facilitate the supply chain attacks to given *Apps*. In our exploration, we defined prompt templates to take in the various vulnerability-relevant information we manually collected, and generated prompts from those templates to query ChatGPT for security test generation. We found that ChatGPT-generated tests demonstrated 24 evidence or proof of vulnerability for 49 Apps. To assess the consistency of test generation, we also evaluated another five state-of-the-art LLMs. All the models generated security tests for at least 17 cases that successfully demonstrate the vulnerabilities. We filed six reports for the newly revealed vulnerabilities in Apps, and got four Common Vulnerability Entries (CVEs) assigned. Our use of ChatGPT outperformed two state-of-the-art security test generators (TRANSFER and SIEGE), by generating a lot more tests and achieving more attacks. Our research will shed light on new research in security test generation.

***Index Terms*—ChatGPT-4.0, supply chain attack, test generation, prompt design, proof of vulnerability, empirical**

I. Introduction

Software development relies on open-source external dependencies and third-party APIs to accelerate development. Developers often integrate these APIs without fully vetting their security vulnerabilities [78], [96], Recent studies showed that many APIs contain known security flaws [61], [98]. When software applications invoke vulnerable APIs without properly validating their security properties, vulnerabilities can get propagated from APIs to those applications [5], [69]. As shown in Fig. 1, attackers can inject or identify exploitable vulnerabilities in third-party libraries through either (1) contributing code to open-source libraries, (2) direct code inspection of the open-source libraries, or (3) consulting publicly available vulnerability databases. They can further exploit these vulnerabilities to propagate attacks through software supply chains and compromise applications built on these libraries. Supply chain attacks (e.g., attack though log4shell vulnerability [6]) grew more than 600%, and caused 12,000 incidents [4] since 2021. Open Web Application Security Project (OWASP) [34] listed "vulnerable and outdated software components" as the sixth top vulnerability.

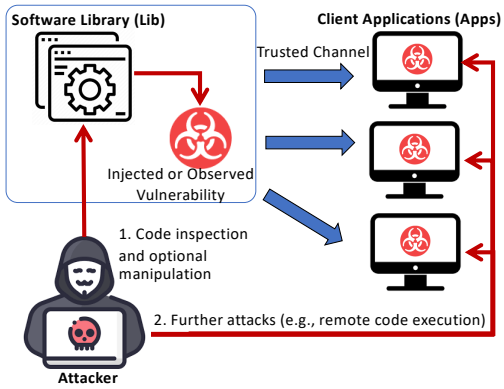

Fig. 1: The threat model of supply chain attacks

To mitigate supply chain attacks, people created tools to identify vulnerable library dependencies in software applications [1], [2], [7], [30], [36], [37], [42], [88], [91], and even suggest fixes for those vulnerabilities [7], [14], [30], [83]. For instance, snyk-test [42] and npm-audit [30] are CLI commands that scan JavaScript (JS) applications for their package dependencies, compare those packages against the package lists in predefined vulnerability databases (e.g., CVE), and report a vulnerability for each found match. However, none of these tools demonstrate how identified vulnerabilities translate to concrete exploits in the developer's specific App context (e.g., denial of service) [66], [105]. For instance, Zhang et al. [105] sent vulnerability reports together with patching suggestions to developers. Some developers did not trust the reports or their security implications; they demanded *proof-of-concept attacks* to demonstrate the security exploit. Such delayed updates can leave real vulnerabilities unaddressed, putting applications at continued security risk.

To help developers assess the impact of reported vulnerabilities and prioritize their fixing process, this paper presents our novel research of generating security tests using LLM, for software applications (Apps) with vulnerable library de-

Y. Zhang, W. Song, Z. Ji, D. Yao, and N. Meng are with the Department of Computer Science, Virginia Tech, Blacksburg, VA 24060.
E-mail: {yingzhang, wenjia7, zhengjie, danfeng, nm8247}@vt.edu

pendencies (Libs). Specifically, if an App calls a vulnerable library API, we generated a prompt for ChatGPT using the following information: (1) the API name, (2) the non-private method M inside App that (in)directly calls that vulnerable API, (3) the Java class defining M, (4) the Lib test that shows **evidence or proof of vulnerability**, and (5) the vulnerability entry ID (e.g., CVE entry ID). Here, proof of vulnerability (PoV) test executes Lib in specialized ways to show behavioral differences between the vulnerable and patched versions of Lib.

With that prompt, we queried ChatGPT to create a security test for App, which test mimics Lib test to show proof of vulnerability in App. The generated test executes App to (i) propagate vulnerabilities from Libs to Apps via calls of the API, and (ii) trigger abnormal behaviors of App such as throwing errors or becoming unresponsive to customers' normal requests. When developers run security tests generated in such a way, they can observe vulnerability propagation paths, foresee the serious consequences due to hackers' successful attacks, assess the severity levels, and better decide whether to address those reported vulnerabilities.

The biggest challenge is ensuring that the generated tests (1) execute those vulnerable APIs called by Apps, (2) trigger any problematic behaviors of Apps, and (3) fail when reported vulnerabilities are not fixed. Iannone et al. [64] and Kang et al. [67] created tools to generate tests using EvoSuite [58], the widely used test generation tool. Unfortunately, both tools fail to generate security tests in many cases. They often spend much time producing irrelevant tests but cannot synthesize the specialized test inputs, code, or oracle.

Our initial experience with ChatGPT showed its great potential in generating code to satisfy software requirements. Additionally, many vulnerable open-source libraries were recorded in publicly available security databases [3], [22], [51], [63], [76], [97]. These databases catalog disclosed software vulnerabilities; they detail on the nature of each vulnerability along with the affected library versions. Such a comprehensive documentation can provide a useful context, where we explored the usage of ChatGPT in generating security tests. We investigated the following research questions (RQs) and observed interesting phenomena:

RQ1: *How effectively does ChatGPT generate security tests?* We created a dataset to include (1) 25 Libs and (2) 49 Apps, with each App depending on a vulnerable library version. For each App, we offered ChatGPT a prompt to describe the vulnerability, App context, and a security test from Lib showing that the patched and vulnerable versions differ. With the prompt, we asked ChatGPT to generate a test for App by mimicking the given test. We found that ChatGPT generated tests for all 49 Apps, 24 of which are security tests that successfully demonstrated evidence or proof of vulnerability (PoV).

RQ2: *How does ChatGPT's security test generation performance differ given various types of prompts?* By changing the default design of our prompt template, we fed ChatGPT with different subsets of the descriptive information in the above-mentioned prompts (see RQ1). We observed that all information elements provided important guidance to ChatGPT, while the security test from Lib was the most important. Without security test from Lib provided, none of the generated tests by ChatGPT could successfully demonstrate PoV.

RQ3: *How does ChatGPT compare with existing tools of security test generation?* We applied ChatGPT and two state-of-the-art tools [64], [67] to the same datasets. Surprisingly, ChatGPT outperformed both tools: it was always able to generate tests, and those tests achieved much higher success rates in realizing PoV.

RQ4: *How does ChatGPT work with few-shot prompting?* We further explored few-shot prompting, by offering ChatGPT one or three examples of test-generation tasks. Interestingly, one-shot prompts led to worse results than the default zero-shot prompting, but three-shot prompts led to better results.

RQ5: *How effectively do different LLMs generate security tests?* We explored PoV test generation capabilities across both closed-source and open-source LLMs with the default prompt setting. Interestingly, closed-source models (GPT-4.0, Claude-3.7-sonnet, Gemini-2.5-pro-preview) demonstrated higher initial test compilability than open-source alternatives (Llama3.3-70b, Deepseek-chat-v3). While Llama3.3-70b achieved the highest number of successful PoV demonstrations (23), each model exhibited unique vulnerability demonstration capabilities.

In summary, this paper makes the following contributions:

- **New LLM experimental methodology.** We explored using a large language model for security test generation. We designed novel prompt templates that take in the PoV-related information that we manually collected, evaluated different ways of using ChatGPT, and compared ChatGPT with state-of-the-art security-test generators: TRANSFER [64] and SIEGE [67].
- **New characterization on LLM capabilities.** We observed ChatGPT to work effectively, given prompts that cover the relevant domain knowledge. ChatGPT outperformed state-of-the-art tools that leverage complex program analysis and genetic programming to generate tests. With zero-shot prompts, ChatGPT successfully generated 24 security tests for 49 Apps.
- **New software security contributions.** Some of ChatGPT-generated tests revealed new vulnerabilities, and we published four CVEs: CVE-2023-31441, CVE-2023-37760, CVE-2023-37761, and CVE-2023-43151.
- **New dataset.** We created a dataset of real-world 49 $\langle Lib, App \rangle$ pairs, which covers vulnerabilities from four big categories. We open-sourced it at https://figshare.com/s/73d194ffcf1c103e7943.

II. A Motivating Example

To facilitate discussion, here we introduce a concrete example to show how vulnerabilities in Libs incur security attacks. Bouncy Castle (BC) is a collection of Java APIs used in cryptography [43]. According to CVE-2020-28052 [17], its releases 1.65 and 1.66 have a vulnerability: `OpenBSDBCrypt.checkPassword(String bcryptString, char[] password)` improperly implements password-checking logic, allowing wrong passwords to be accepted as valid ones.

Listing 1 shows the security test defined by a BC version later than 1.66, which demonstrates PoV. Ideally, if a BC version has no vulnerability, the first assertion (line 9) succeeds as the first parameter `tokenString` was derived from the password `test-token`; the second assertion (line 12) succeeds as the first parameter `tokenString` was not from `wrong-token`. However, BC 1.65 and BC 1.66 fail the second assertion, as the invalid password `wrong-token` is wrongly considered to match `tokenString`. Such a security test demonstrates the problematic behaviors of vulnerable library versions, and implies the potential of security exploits (e.g., sending in wrong passwords to pass identity authentication).

Although Listing 1 shows a PoV of the library, it does not show how Apps built on top of vulnerable BC versions can behave abnormally or get attacked. Existing vulnerability detectors can report such vulnerable API calls in Apps [68], [90], [105], [106]. However, as they do not show how vulnerable API calls introduce vulnerabilities or induce supply chain attacks to Apps, App developers are reluctant to revise Lib usage accordingly.

In this project, we explore to mitigate supply chain attacks, by generating security tests to simulate hackers' potential ways of using Apps maliciously. Our approach is using ChatGPT to generate security tests for Apps to mimic Lib tests. **Our goal is to investigate how effectively ChatGPT generates App-specific security tests, when it is given relevant information about the Lib vulnerabilities, App context, and exemplar tests.** Thus, we assume some tools or domain experts to detect potential vulnerabilities and provide relevant data to ChatGPT for test generation. Once a test is generated successfully, domain experts or developers can run App with that test, observe the problematic program behaviors themselves, assess the security implication of supply chain attacks, and decide whether to eliminate vulnerabilities by upgrading library versions or replacing API calls.

III. Methodology

We conducted an empirical study on ChatGPT's capability of generating PoV, for Apps that depend on vulnerable Libs and call vulnerable APIs. We chose to experiment with ChatGPT-4.0, because our preliminary work shows that ChatGPT has the best performance across the different LLMs as shown in Section IV-F.

```
1  blic void performTest() throws Exception {
2  ... ...
3  int costFactor = 4;
4  SecureRandom random = new SecureRandom();
5  salt = new byte[16];
6  for (int i = 0; i < 1000; i++) {
7    random.nextBytes(salt);
8    final String tokenString =
     ↪  OpenBSDBCrypt.generate("test-token".toCharArray(),
     ↪  salt, costFactor);
9    isTrue(OpenBSDBCrypt.checkPassword(tokenString,
     ↪  "test-token".toCharArray()));
10   /* A safe BC version passes the following assertion; a
     ↪  vulnerable one fails the assertion, as it considers
     ↪  unmatched passwords to match. */
11   isTrue(!OpenBSDBCrypt.checkPassword(tokenString,
     ↪  "wrong-token".toCharArray()));
12 } }
```

Listing 1: A Lib test to show PoV in Lib

Our study has three phases: dataset construction, prompt design, and result validation. Phase I (Section III-A) collects known vulnerabilities in Libs, Lib tests showing PoV, and Apps that can be affected by their calls of vulnerable APIs. Phase II (Section III-B) adopts the information collected by Phase I, formulates a variety of prompts for individual $\langle Lib, App \rangle$ pairs, and sends prompts to ChatGPT. These prompts ask ChatGPT to leverage all information provided, to generate security tests for Apps. Phase III (Section III-C) gathers all outputs by ChatGPT, assesses the quality of generated tests, and evaluates ChatGPT's capability accordingly.

A. Phase I: Dataset Construction

While existing datasets [64], [67] offer potential benchmarks for evaluating security test generation, we found them inadequate for our specific assessment of ChatGPT. One of the datasets is not quite usable while the other is unrepresentative, so we decided to create a new dataset. Specifically, although the dataset used to evaluate TRANSFER [67] covers 22 Libs and 42 Apps, 21 of the Apps could not easily run to demonstrate vulnerability PoV due to (1) project removal from GitHub, (2) compilation or dependency issues, (3) missing vulnerable API calls, and (4) unconvincing vulnerabilities in test files that are excluded from software deliverables. The dataset used to evaluate SIEGE [64] includes 11 Libs and 11 Apps, where Apps are handcrafted toy projects instead of real-world programs. In order to systematically evaluate the test generated by ChatGPT, we took two steps to create a dataset: (1) finding vulnerabilities with exemplar PoV tests, and (2) getting vulnerable Libs as well as dependent client Apps.

1) Locating Vulnerabilities with Exemplar PoV Tests: Vulnerabilities sparsely exist in software libraries. To efficiently locate vulnerabilities, we started with the datasets mentioned by prior work [67], [89] and initiated our exploration with 628 entries. Each entry is linked to a CVE entry or JIRA issue, describing a vulnerable Lib and a GitHub repository showing both the vulnerable and patched versions of Lib.

Our initial step involved automatically filtering the commit history to identify commits adding test files. The first and fifth authors then perform an in-depth analysis to identify the vulnerable API for each of these filtered commits. As vulnerability descriptions may not always precisely pinpoint vulnerable library APIs, the authors spend time understanding the program context and commit details to accurately identify these APIs. They consider the APIs is vulnerable if it (1) is mentioned or implied by the vulnerability description of CVE or JIRA entry and gets revised, (2) directly or indirectly calls the described vulnerable API, (3) is invoked by the described API and is the root cause for the described vulnerability, or (4) shares the same root cause with the described API (i.e., they both call the same root-cause vulnerable method). To determine which API is the root cause, the first and fifth authors discussed each case together, cross-validating their findings to ensure consistency and reliability. To extract the PoV test, we further chose vulnerability entries based on two criteria:

(a) Exemplar Security Test: Lib has at least one JUnit test from the patched version, to demonstrate behavioral differences between the vulnerable and patched versions.
(b) Successful Execution: The patched version of Lib should run smoothly with the security test, requiring no extra manual effort for bug fixing or software (re)configuration.

Criterion (a) ensures the exploitability of confirmed vulnerabilities. Namely, if no Lib test is available to demonstrate PoV, it is hard for us to manually craft and justify the ground truth of test generation. Criteria (b) ensures that we can run the security test defined for Lib, to observe the behavioral differences between vulnerable and patched versions. In our study, we implemented the two criteria as filters to refine initial vulnerability datasets. The filters separately removed 427 and 156 entries, leaving 45 for further processing.

2) Collecting Libs and Apps for Vulnerabilities: For each library identified in the 45 refined entries (see Section III-A1), We automatically crawled GitHub using the GitHub API to find client applications depending on these libraries. We use the library or package name as keywords, filtering for projects whose dependency versions fall within the vulnerable range, and limiting our crawling to the first 10 pages of results, given the large volume of projects returned. The first, second, and third authors then manually inspected the code of these projects to verify actual usage of the vulnerable functionality (beyond mere dependency inclusion), with the goal of identifying up to four client applications per library that satisfy both criteria (c) and (d):

(c) At least one non-private Java method (not test) in App directly or indirectly calls vulnerable API(s) in Lib, and gets impacted by the library vulnerability.
(d) App compiles and runs successfully.

Criterion (c) ensures the feasibility of creating PoV tests. Basically, if users craft malicious input values to feed certain public or protected method(s) in App (i.e., the callers of vulnerable APIs), they can run vulnerable APIs in malicious ways and thus realize attacks. Criterion (d) ensures that we can check the correctness of tool-generated tests via compilation and program execution. This filtering process removed 16 from the 45 entries mentioned above, because we found no client project to satisfy both criteria for those entries. As shown in Table I, our dataset includes 29 vulnerability entries, corresponding to 25 unique Libs. These libraries cover various domains, such as data processing (e.g., Apache Commons Codec [15]), web development (e.g., Apache CXF [11]), and security (e.g., Spring Security [39]). Most Libs have a single vulnerability (e.g., Dom4j [18]), while a few have multiple (e.g., XStream [46]). In Table I, we identified 4 major categories among the 29 vulnerabilities: denial of service, directory traversal, remote code execution, and others. **Affected Library Versions** shows the vulnerable library versions described by each CVE entry or JIRA issue. **Vulnerable API(s) & Potential Exploit** shows the vulnerable APIs and security consequence we summarized by inspecting all relevant data.

According to our experience, it is very challenging to identify a sufficient number of Apps satisfying (c)–(d) for any vulnerable library. To conduct a representative empirical study with sufficient data points, we extensively explored the version history of the retrieved Apps even though they did not depend on vulnerable library versions in the current version. Specifically for each found project *App* satisfying criteria (c)–(d), we examined the version history to determine whether any earlier version, denoted as App_i, depends on a vulnerable library version. If App_i exists, we checked out App_i to prepare for ChatGPT usage. Otherwise, we manually revised the dependency version. For instance, OpenRefine [31] is a library whose versions before 3.2-beta suffer from CVE-2018-19859. However, we only found one client project for it, which depends on a safe version of OpenRefine (i.e., 3.3). To make sure that this client is still usable in our study, we downgraded the library dependency in the configuration file to 3.1 without further modification. This strategy reflects a common real-world scenario where Apps rely on outdated dependencies due to delayed patching. We do not inject handcrafted vulnerable logic into Apps, as the approach risks introducing artificial or irrelevant behaviors and was therefore avoided. By contrast, our revision preserved the application code logic and only altered the dependency version to formulate realistic conditions. Our manual revision of dependencies does not compromise the validity of our research, as we fairly compared all approaches of security test generation on the same set of client Apps, no matter whether their vulnerable dependencies are real or injected. The last column in Table I shows the number of clients we included for each Lib. There are five projects with injected vulnerable dependencies, all of which are marked with asterisks (*). At the end of Phase I, our dataset consists of all relevant information for 49 $\langle App, Lib \rangle$ pairs.

Our newly created dataset overlaps with TRANSFER's dataset [67] by sharing 17 Apps in common, while sharing 0 App with SIEGE's dataset [64]. This fact implies that our dataset is very different from existing ones; it can enrich the knowledge body of exploitable vulnerabilities in Apps. Furthermore, we classified the 49 functions-under-test in our dataset into 3 categories:

- C1: Functions directly calling vulnerable APIs and sharing the same parameter lists with called APIs. For instance, if a client function $p(int, int)$ is defined to call API $q(int, int)$, the client function belongs to C1.
- C2: Functions directly calling vulnerable APIs but defining different parameter lists from called APIs. For instance, if a client function $p(long)$ is defined to call API $q(int, int)$, then the client function belongs to C2.
- C3: Functions indirectly calling vulnerable APIs. For instance, if a client public function $p()$ is defined to call API $q()$ in the following manner: $p() \rightarrow r() \rightarrow q()$, where $r()$ is a private client function directly calling $q()$, then $p()$ belongs to C3.

C1–C3 separately cover 20, 20, and 9 functions. We hypothesize the complexity comparison of test-generation tasks among these categories to be C1 $<$ C2 $<$ C3, mainly because the less commonality is shared between functions-to-test and vulnerable APIs, the more difficult it is for ChatGPT to generate tests.

TABLE I: The library vulnerabilities and client applications included in our dataset

Category	Vulnerability Entry ID	Library	Affected Library Versions	Vulnerable API(s) & Potential Exploit	# of Apps
Denial of Service (13)	CVE-2017-7957 (CWE-20)	XStream [46]	[, 1.4.9]	`XStream.fromXML(...)` mishandles attempts to create an instance of the primitive type "`void`" during unmarshalling, leading to a remote application crash, i.e., denial of service (**DoS**).	2
	CVE-2018-1000873 (CWE-20)	Jackson-Modules-Java8 [21]	[, 2.9.8)	`ObjectMapper.readValue(...)` triggers DoS when it deserializes a very large decimal value to time.	3
	CVE-2018-11761 (CWE-611)	Apache Tika [13]	[0.1, 1.18]	`SAXParser.parse(...)` was not configured to limit entity expansion, and thus could lead to DoS.	1
	CVE-2018-12418 (CWE-835)	Junrar [26]	[,1.0.1)	The `Archive` constructor gets into an infinite loop when handling corrupt RAR files.	1
	CVE-2018-1274 (CWE-770)	Spring Data Commons [38]	[1.13, 1.13.10], [2.0, 2.0.5]	`PropertyPath.from(...)` allocates resource without limits, and thus can cause DoS due to its consumption of CPU and memory.	1
	CVE-2019-10093 (CWE-770)	Apache Tika	[1.19, 1.21]	`Parser.parse(...)` enables a carefully crafted 2003ml or 2006ml file to consume all available SAXParsers in the pool.	1
	CVE-2019-12402 (CWE-835)	Apache Commons Compress [16]	[1.15, 1.18]	Malicious inputs to `ZipArchiveOutputStream.putArchiveEntry(...)` or `ZipEncoding.encode(...)` can cause infinite loops.	1
	CVE-2020-28491 (CWE-770)	Jackson Dataformat: CBOR [20]	[, 2.11.4), (2.12.0-rc1, 2.12.1)	`ObjectMapper.createParser(...)` allocates resources without limits; it can cause `java.lang.OutOfMemoryError`.	1
	CVE-2021-27568 (CWE-754)	Json-smart [28], [29]	v1:[, 1.3.2), v2: [, 2.3.1), [2.4, 2.4.1)	`JSONParser.parse(...)` throws an uncaught exception, which can cause an application crash or expose sensitive information.	2
	CVE-2021-30468 (CWE-835)	Apache CXF [11]	[, 3.3.11), [3.4.0, 3.4.4)	Malicious inputs to `JsonMapObjectReaderWriter.fromJson(...)` or `JsonMapObjectReaderWriter.fromJsonToJsonObject(...)` can result in an infinite loop.	1
	CVE-2022-45688 (CWE-787)	JSON-java(i.e., hutool-json) [41]	[, 20230227)	Malicious inputs to `XML.toJSONObject(...)` or `JSONML.toJSONObject(...)` can trigger DoS.	3
	TwelveMonkeys-595	TwelveMonkeys [23]	[0, 3.6.4)	A corrupt JPEG file to `ImageReader.read(...)` can cause DoS.	2 (1*)
	Zip4j-263	Zip4j [40]	[0, 2.7.0)	The `ZipFile(...)` constructor can take in a null File reference, which later produces a null pointer exception.	2
Directory Traversal (5)	CVE-2018-1002200 (CWE-22)	Plexus Archiver [35]	[,3.6.0)	`UnArchiver.extract(...)`, `ZipUnArchiver.extract(...)`, and `TarGZipUnArchiver.extract(...)` allow attackers to write to arbitrary files via "../" in an archive entry (**Zip Slip**).	2
	CVE-2018-1002201 (CWE-22)	ZT Zip [47]	[, 1.13)	`ZipUtil.unpack(...)` allows attackers to write to arbitrary files via archive extraction (Zip Slip).	1
	CVE-2018-19859 (CWE-22)	OpenRefine [31]	[, 3.2-beta)	`ImportingUtilities.allocateFile(...)` allows arbitrary file write via archive extraction (Zip Slip).	1(1*)
	CVE-2021-29425 (CWE-20)	Apache Commons IO [10]	[, 2.7)	`FileNameUtils.normalize(...)` enables **directory traversal**, which provides access to files beyond the target file location.	3
	HTTPCLIENT-1803	Apache HttpClient	[,4.5.3)	The `URIBuider` constructor, `URIBuilder.setHost(...)`, `URIBuilder.build(...)`, and `URIBuilder.toString(...)` can result in directory traversal.	1
Remote Code Execution (4)	CVE-2017-7525 (CWE-22)	Jackson Databind [19]	[, 2.6.7.1) [2.7.0, 2.7.9.1) [2.8.0, 2.8.9)	A deserialization flaw in the library allows maliciously crafted inputs to `ObjectMapper.readValue(...)` to trigger remote code execution.	2
	CVE-2020-26217 (CWE-78)	XStream	[, 1.4.14)	Malicious inputs to `XStream.fromXML(...)` allow attackers to run arbitrary shell commands.	3
	CVE-2021-23899 (CWE-611)	OWASP JSON Sanitizer	[,1.2.2)	`JsonSanitizer.sanitize(...)` may allow hackers to inject arbitrary code into embedding documents.	1
	CVE-2022-25845 (CWE-502)	Fastjson [8]	[, 1.2.83)	`JSON.parseObject(...)` may deserialize untrusted data, allowing hackers to attack remote servers.	2
Others (7)	CODEC-134	Apache Commons Codec [15]	[, 1.13)	Malicious inputs to `Base64.decodeBase64(...)` or `Base64.decode(...)` can realize **covert channel** [103], which creates a capability of transferring data between processes that should not communicate	3
	CVE-2018-1000632 (CWE-91)	Dom4j [18]	[, 2.1.1)	Malicious inputs to `DocumentHelper.createElement(...)` or `Branch.addElement(...)` can result in **XML injection**, which tampers with XML documents.	2
	CVE-2020-13956	Apache HttpClient [12]	[, 4.5.13,), [5.0.0, 5.0.3)	Malicious inputs to `CloseableHttpClient.execute(...)` or `URIUtils.extractHost(...)` trigger **Blind Server-Side Request Forgery (SSRF)**, which attack induces an application to issue a back-end HTTP request to a supplied URL, but the response from the back-end request is not returned to the application's front-end response.	1
	CVE-2020-13973 (CWE-20)	OWASP JSON Sanitizer [33]	[,1.2.1)	`JsonSanitizer.sanitize(...)` does not properly escape disallowed characters, and thus facilitates **cross-site scripting (XSS)**, which enables the browser to unknowingly execute malicious script on the client side and perform actions that are otherwise blocked by the browser's Same Origin Policy.	1
	CVE-2020-28052 (CWE-20)	Bouncy Castle	1.65, 1.66	`OpenBSDBCrypt.checkPassword(...)` improperly verifies passwords, allowing wrong ones to be accepted as valid ones.	2 (1*)
	CVE-2020-5408 (CWE-20)	Spring Security [39]	[4.2.0, 4.2.16), [5.0.0, 5.0.16), [5.1.0, 5.1.10), [5.2.0, 5.2.4), [5.3.0, 5.3.2)	`BCryptPasswordEncoder.encode(...)` presents cryptographic weakness, which may allow hackers to decrypt encrypted messages via a dictionary attack.	2 (2*)
	CVE-2023-34454 (CWE-20)	snappy-java [45]	[, 1.1.10.1)	`Snappy.compress(...)` improperly validates array length, and may cause Access Violation errors.	1

* indicates the number of clients with injected vulnerable dependencies.

Prompt Design

Write a JUnit test for function: *[function-name]* in "CLIENT CODE". The test should verify whether the function *[function-name]* is affected by the vulnerability: *[vulnerability-ID]*, or affected by vulnerable APIs: *[vulnerable-API-list]*. Mimic the test function "TEST FUNCTION:" *[test-function-name]* I provide to generate the input in your test. For the provided code "CLIENT CODE", mock the classes and functions that are not Java SE APIs and have no definition provided.

```
TEST FUNCTION:
``` Java
[exemplar-test-impl]
```

CLIENT CODE:
``` Java
[client-code-impl]
```
```

Fig. 2: Our default prompt template

Prompt Example

Write a JUnit test for function: *[check]* in "CLIENT CODE". The test should verify whether the function *[check]* is affected by the vulnerability: *[CVE-2020-28052]*, or affected by vulnerable APIs: *[OpenBSDBCrypt.checkPassword]*. Mimic the test function "TEST FUNCTION:" *[performTest]* I provide to generate the input in your test. For the provided code "CLIENT CODE", mock the classes and functions that are not Java SE APIs and have no definition provided.

```
TEST FUNCTION:
``` Java
// Here we omit details of BC security test
// (Listing 1) for brevity.
@Test
public void performTest() throws Exception { ... }
```

CLIENT CODE:
``` Java
// Here we omit details of the class, which
// defines the method check (...) in Listing 2
public class BcryptPasswordHashFunction
    implements PasswordHashFunction { ... }
```
```

Fig. 3: A prompt derived from the default template

B. Phase II: Prompt Design

With the information collected in Phase I, we formulated prompts and asked ChatGPT to generate tests. This section first introduces the information elements extracted from our dataset for each $\langle App, Lib \rangle$ pair (Section III-B1). It then introduces our various ways of constructing ChatGPT prompts using those elements (Sections III-B2 and III-B3).

1) Information Elements Extracted: For each $\langle App, Lib \rangle$ pair, we extracted seven elements for later prompt creation.

- **(i)** A GitHub project of the client application App.
- **(ii)** A vulnerable version of Lib on which App depends.
- **(iii)** The vulnerable Lib API(s) called by App.
- **(iv)** The non-private method M inside App that directly or indirectly calls vulnerable APIs.
- **(v)** The Java class C defining method M.
- **(vi)** The Lib test T (i.e., a Java method). If it indirectly calls vulnerable APIs, the definition of all methods standing between T and APIs is also included.
- **(vii)** The vulnerability entry ID (i.e., CVE or JIRA entry ID).

Elements (i)–(ii) are essential to represent an $\langle App, Lib \rangle$ pair; (iii) and (vii) describe the library vulnerability; (vi) shows an exemplar PoV test for Lib, and (iv)–(v) present relevant context in App. These seven elements cover all necessary information we can recognize to describe a generation task of PoV test for App.

2) The Design of Our Default Prompt Template: We believe that given more comprehensive prompts, ChatGPT is likely to generate better tests. Therefore, we designed a default prompt template to cover all seven elements mentioned above. As shown in Fig. 2, the template has seven variables or customizable parameters to accept data for each task-specifying prompt. In more detail, variable *[function-name]* is the name of M (see **(iv)**); *[vulnerability-ID]* refers to **(vii)**, *[vulnerable-API-List]* refers to **(iii)**; *[test-function-name]* is the name of example test (see **(vi)**); *[exemplar-test-impl]* refers to **(vi)**; and *[client-code-impl]* refers to **(v)**.

Overall, the template asks ChatGPT to generate a JUnit test for a function in App, so that (1) the test verifies whether that App function is affected by a known vulnerability or by specified vulnerable APIs, and (2) the test mimics the exemplar test to generate malicious inputs. With this template, we generated 49 prompts.

For the motivating example described in Section II, we actually found a GitHub project depending on Bouncy Castle. As shown in Listing 2, the project defines a method `BcryptPasswordHashFunction.check(...)` to directly call vulnerable API `OpenBSDBCrypt.checkPassword(...)`. Although both the API and its caller take in two parameters, the caller method `check(...)` has to convert its second parameter `password` before calling that API. To generate a security test for `check(...)`, we formulated the prompt shown in Fig. 3 by customizing our template. Fig. 3 only shows partial code of the example test and client class to simplify our presentation, while the actual prompt we sent to ChatGPT includes the complete code.

Given the prompt, ChatGPT successfully generated an executable security test as requested. Listing 3 shows a brief version of the generated code: a class named `BcryptPasswordHashFunctionTest` defines a test function `testCheckFunction()`, to call the vulnerable API with appropriate formats of the seed inputs of exemplar test. The generated test is very similar to the exemplar test, but it shows PoV for client code.

3) The Design of Alternative Prompt Templates: In addition to the default prompt template, we also defined five variant templates by removing a single element from (iii)–(vii) each time. In this way, we can explore how different information

```
1 blic class BcryptPasswordHashFunction implements
  ↪ PasswordHashFunction {
2 ...
3 // check(...) calls the vulnerable API, so it can be
  ↪ affected by CVE-2020-28052.
4 @Override
5 public boolean check(String passwordHash, String password) {
6   return OpenBSDBCrypt.checkPassword(passwordHash,
    ↪ password.toCharArray());
7 } ... }
```

Listing 2: A GitHub project defines a Java file to call `OpenBSDBCrypt.checkPassword(String, char[])`

TABLE II: The six prompt templates we explored

Id	Prompt Template	Id	Prompt Template
P	Default (including iii-vii)	P_3	Without C (v)
P_1	Without vulnerable APIs (iii)	P_4	Without the exemplar test (vi)
P_2	Without M (iv)	P_5	Without the vulnerability ID (vii)

elements contribute to ChatGPT's effectiveness. As shown in Table II, to facilitate presentation, we use P to refer to the default template, and use P_1–P_5 to refer to the five variants. For instance, P_5 leaves out (vii), but takes in (iii)–(vi). We then generated 49 prompts using each variant template.

C. Phase III: Result Validation

After sending all prompts to ChatGPT, we gathered and recorded ChatGPT's outputs. We integrated the generated tests into Apps using the following rules: if a generated test class C' has a unique name and defines one or more test functions for method-to-test M, we place the class into the test folder. Otherwise, if C' has the same name as an existing class, we put the non-conflicting content into the existing class; if C' defines test functions for a non-public method M, we placed the tests into any existing (test) class where M is accessible.

For each generated test, we compiled App with the test to check for compilation errors; if some compilation errors were obvious and easy to fix (e.g., missing/wrong package names or missing library dependencies), we fixed those errors manually to explore whether ChatGPT could synthesize the most important logic of security tests. After one or multiple iterations of the compilation-and-fixing procedure, if a test compiled successfully, we further executed App with that test to observe runtime behaviors. If any exception or runtime error was thrown, we studied the exception/error message, inspected the intermediate program status via step-by-step debugging, and discussed the relevance with PoV among authors until reaching a consensus.

IV. Experiments and Results

We empirically investigated three research questions (RQs):

RQ1: *How effectively does ChatGPT generate security tests?* We explored the strengths and weaknesses of ChatGPT in security test generation.

RQ2: *How does ChatGPT's security test generation performance differ given various types of prompts?* Among the information elements (iii)–(vii), we explored which one is more crucial.

RQ3: *How does ChatGPT compare with existing tools of security test generation?* We wanted to learn how well LLM-based security test generation compares with existing tools based on program analysis.

RQ4: *How does ChatGPT work with few-shot prompting?* We explored whether ChatGPT works better when one or more examples of test-generation tasks are provided.

RQ5: *How effectively do different LLMs in generate security test?* We investigated whether our findings generalize across different LLMs by evaluating the performance of five state-of-the-art models on the security test generation tasks.

```
1  . // We omit less important details for brevity
2  blic class BcryptPasswordHashFunctionTest {
3  ...
4  @Test
5  public void testCheckFunction() {
6    int costFactor = 4;
7    for (int i = 0; i < 1000; i++) {
8      random.nextBytes(salt);
9      final String tokenString =
       ↪  OpenBSDBCrypt.generate("test-token".toCharArray(),
       ↪  salt, costFactor);
10     assertTrue(bcryptPasswordHashFunction.check(
       ↪  tokenString, "test-token"));
11     /* The App should fail the following assertion, when it
       ↪  depends on a vulnerable BC version that messes up
       ↪  correct with incorrect passwords.*/
12     assertFalse(bcryptPasswordHashFunction.check(
       ↪  tokenString, "wrong-token"));
13 } }
```

Listing 3: A brief and commented version of the security test successfully generated by ChatGPT.

This section first introduces the metrics we defined to assess tools of security test generation (Section IV-A). It then explains our experiments and results for RQs.

A. Metrics

There are three metrics used in our experiments:

Tool Applicability (A) counts for how many $\langle App, Lib \rangle$ pairs, a tool generates a security test.

Test Compilability (C) counts the number of generated tests that are compilable, with minor fixes applied.

PoV Demonstration (D) counts the number of compilable tests that successfully demonstrate PoV for the known vulnerabilities as expected.

B. ChatGPT's Effectiveness in Security Test Generation (RQ1)

We created 49 prompts using the default prompt template P, and sent them to ChatGPT , which uses GPT-4.0 as an underlying model in our experiment.

As shown in Table III, ChatGPT generated tests for all prompts. Of these, 26 tests are compilable and runnable as is. In contrast, 12 tests become compilable and runnable after we manually apply minor fixes, such as adding missing dependencies, handling exceptions, or correcting hardcoded file paths. 24 (including five tests with minor fixes) of these 38 tests effectively mimic the behaviors of given library tests, and successfully demonstrate PoV by throwing relevant errors or runtime exceptions.

1) Uncompilable Tests.: Among the 49 generated tests, 11 tests do not compile and cannot get easily fixed via minor changes. Specifically, two of the tests violate Java access rules, such as directly accessing private members outside of declaring classes. Five tests use undefined program entities (e.g., methods and classes). Another four tests call methods inappropriately, by missing some parameters or using the wrong type of parameters. Our observations imply that ChatGPT does not guarantee code compilation, even though the majority of tests it generated (38/49) are easy to compile.

2) Less Effective Tests.: 14 generated tests do not trigger vulnerabilities as expected. They either throw exceptions/errors other than the expected ones, or throw no exception/error at all. Three reasons can explain such ineffectiveness. First, Mockito [27]—a mocking framework—was used by ChatGPT to mock unknown variables, methods, or classes

TABLE III: Security tests generated by ChatGPT (Tool Applicability (A): 49, Test Compilability (C): 38, PoV Demonstration (D) 24)

Idx	Vulnerability Entry ID	# of Clients	A	C	D	Idx	Vulnerability Entry ID	# of Clients	A	C	D
1	CODEC-134	3	3	2	2	15	CVE-2020-13973	1	1	0	0
2	CVE-2017-7525	2	2	2	2	16	CVE-2020-26217	3	3	3	2
3	CVE-2017-7957	2	2	1	1	17	CVE-2020-28052	2	2	2	1
4	CVE-2018-1000632	2	2	1	1	18	CVE-2020-28491	1	1	1	0
5	CVE-2018-1000873	3	3	2	0	19	CVE-2020-5408	2	2	2	2*
6	CVE-2018-1002200	2	2	1	1	20	CVE-2021-23899	1	1	1	0
7	CVE-2018-1002201	1	1	1	1	21	CVE-2021-27568	2	2	2	2
8	CVE-2018-11761	1	1	0	0	22	CVE-2021-29425	3	3	2	2
9	CVE-2018-12418	1	1	0	0	23	CVE-2021-30468	1	1	1	0
10	CVE-2018-1274	1	1	0	0	24	CVE-2022-25845	2	2	2	2
11	CVE-2018-19859	1	1	1	0	25	CVE-2022-45688	3	3	3	1
12	CVE-2019-10093	1	1	1	0	26	CVE-2023-34454	1	1	1	1
13	CVE-2019-12402	1	1	0	0	27	HTTPCLIENT-1803	1	1	1	0
14	CVE-2020-13956	1	1	1	1	28	TwelveMonkeys-595	2	2	2	0
						29	Zip4j-263	2	2	2	2

*indicates the number of clients with injected vulnerable dependencies

when generating tests. Unfortunately, the framework could not mock everything (e.g., final classes), and sometimes led to MockitoExceptions. Second, in generated tests, the parameters passed to M are not always well prepared. They may be malformed, include `null`-values in critical fields, or fail to contain the essential values to trigger vulnerabilities. Third, our generated tests uncovered two unexpected bugs. Specifically, one test exposed a concurrency issue in `NCI-Agency/anet's sanitizeJson()`, leading to a potential denial-of-service vulnerability in applications. This issue has been reported and assigned CVE-2023-31441.The other bug initially presented as a `NullPointerException`. However, upon inspecting the exception chain, we discovered that the test revealed a Server-Side Request Forgery (SSRF) vulnerability within the AssetVersionTransformer component of the `aem-caching` library (version 0.0.3-SNAPSHOT).

3) Effective Tests.: Twenty-four generated tests can trigger vulnerabilities as expected, including two cases involving projects with injected vulnerable dependencies. For these tests, ChatGPT successfully extracted vulnerability-triggering inputs from the exemplar tests, reused those inputs to call method M somehow, and presented relevant abnormal program behaviors (e.g., infinite loop). As most vulnerabilities were already fixed by the latest versions of subject projects, we only filed reports for four of the newly revealed vulnerabilities in Apps. Two of them were approved: CVE-2023-37760 and CVE-2023-43151.

Based on our experience, when M calls vulnerable API(s) directly and defines a parameter list the same as the called API(s), ChatGPT was more likely to succeed in producing security tests; 15 of the 24 tasks included such M (i.e., methods under testing). This is understandable as ChatGPT does not reason about program logic. When it tries to generate a test similar to a given test, any commonality among the App context, vulnerable API, and exemplar test can facilitate knowledge reuse and test mimicry.

Meanwhile, when M calls vulnerable API(s) indirectly or defines a different parameter list from the called API(s), ChatGPT was less capable because less commonality is shared between App and Lib tests. Interestingly, among the 24 tasks well handled by ChatGPT, 9 tasks involve M with a different parameter list and 1 task defines M to indirectly call the vulnerable API. It implies three things. First, ChatGPT is promising to effectively generate tests, even though the vulnerability propagation path from Lib to App is more complex or less obvious. Second, ChatGPT's effectiveness decreases as the test-generation tasks become more challenging. For the three categories of functions (i.e., C1–C3) in our dataset, ChatGPT's success rates of demonstrating PoV are 75% (15/20), 45% (9/20), and 11% (1/9). Namely, ChatGPT is more effective in generating tests when M shares more commonality with vulnerable APIs, but less effective when there is less commonality. Third, our experiment results are generalizable, as our dataset is not limited to trivial cases where functions-under-test and vulnerable APIs share both parameter lists and program context. Namely, the more commonality is shared between M and vulnerable APIs (category C1), the more likely ChatGPT can generate security tests successfully.

Finding 1: *ChatGPT is promising in generating security tests for known library vulnerabilities. Given 49 test generation tasks, it produced 49 tests, 38 of which are easy to compile and 24 tests successfully demonstrated PoV.*

For the 29 vulnerabilities we investigated, the 29 exemplar tests have 5 types of test oracles: thrown exceptions, thrown runtime errors, timeouts, infinite loops, and expected return-values of function calls. Among those exemplar tests, when vulnerable functions are called, 2 tests throw expected exceptions; 10 tests do not throw exceptions as expected; 4 tests throw runtime errors; 1 test does not throw an error as expected; 3 tests encounter time out; 4 tests run into infinite loops; 2 tests do not output expected values; 3 tests produce the expected problematic outputs.

Among the 24 successfully generated tests, 22 of them follow similar vulnerability exploitation logic as the given exemplar tests. In particular, when vulnerable functions are called, 3 of the generated tests throw expected exceptions; 13 tests do not throw exceptions as expected; 3 tests throw runtime errors, 1 test does not throw an error as expected; 1 test encounters time out; 2 tests do not output expected values; 1 test produces the expected problematic output. None of the generated tests run into infinite loops, which implies that it may be harder for ChatGPT to generate tests to demonstrate

TABLE IV: The comparison of tests generated in different ways

Id	Prompt Template	Tool Applicability	Test Compilable?				PoV Demonstrated?			
			Yes	No			Yes	No		
		(A)	(C)	Access Rule Violation	Incorrect Method Calls	Unknown Entity Usage	(D)	No error/ exception	Mockito exception	Other exceptions/ errors
P	Default (all elements)	49	38	2	4	5	24	3	2	9
P_1	Without vulnerable APIs (iii)	49	34	3	2	10	15	6	8	5
P_2	Without M (iv)	49	39	4	1	5	14	3	8	14
P_3	Without C (v)	49	15	3	1	30	1	7	4	3
P_4	Without the exemplar test (vi)	49	30	4	2	13	0	11	11	8
P_5	Without the vulnerability ID (vii)	49	36	6	0	7	14	3	5	14

such abnormal program behaviors.

Finding 2: *ChatGPT effectively mimics human-crafted tests; 22 of the 24 tests it successfully generated have matching logic with the given exemplar tests.*

To check the consistency of ChatGPT's outputs, we conducted an experiment by sending 25 test-generation tasks 5 times to ChatGPT, using 125 distinct conversation sessions. We randomly sampled those tasks, without bias towards any data. ChatGPT produced consistent results 113 of 125 times (90%). Our results imply that ChatGPT often produces very similar results, given the same prompt multiple times.

C. Impact of Information Elements on ChatGPT's Outputs (RQ2)

We used template variants P_1–P_5 to generate prompts for 49 $\langle App, Lib \rangle$ pairs, and sent all prompts to ChatGPT for results.

1) ChatGPT's Applicability Given Divergent Types of Prompts: As shown in Table IV, no matter what information item in the default template was removed, the resulting prompts always guided ChatGPT to produce tests. It means that ChatGPT has great applicability: it is always applicable no matter how the prompts were formulated.

2) ChatGPT's Test Compilability Given Divergent Types of Prompts: As shown in Table IV, when P_2 was used and M was not specified, ChatGPT generated more compilable tests than what it did for the default template P (39 vs. 38). However, when P_3 and P_4 were used, a lot fewer generated tests compile, i.e., 15 and 30. One possible reason is that both P_3 and P_4 significantly removed the code context, while the other templates removed almost no code context. As a generative AI tool, ChatGPT predicts the next word(s) given a data sequence, by using (1) an encoder to process the input sequence and (2) a decoder to generate the output [25]. It tended to generate more compilable tests when more relevant program context was provided.

Among the 6 prompt templates, P_3 caused ChatGPT to create the most uncompilable tests—34; 30 of these tests fail compilation due to their usage of unknown entities. This may be because P_3 does not specify the Java class C holding the function-to-test; ChatGPT could not identify many valid or usable entities available in the software projects, so it usually refers to some nonexistent entities in the produced tests. In contrast, P_2 caused ChatGPT to create the fewest uncompilable tests—10, only 5 of which fail compilation due to their usage of unknown entities. This comparison implies that ChatGPT could produce more compilable tests when (1) more program context is provided in prompts, and (2) there is no constraint on what method to test.

No matter what template we used, ChatGPT always produced uncompilable tests for some prompts. This observation indicates that ChatGPT does not strictly follow Java rules on syntax or semantics. As a generative AI model, it was trained to predict the next words or phrases to follow a given sequence. Thus, the generated code may violate access rules, call methods with inappropriate parameter lists, or use unknown program entities. This observation implies the necessity of applying sanity checks to ChatGPT-generated code and fixing any revealed bugs, to ensure the program quality.

Finding 3: *Among the five template variants we explored, P_3 (Without the client project class C) and P_4 (Without the exemplar test) caused ChatGPT to work considerably worse in producing compilable tests. ChatGPT tended to produce more compilable tests, given more contextual code and fewer constraints relevant to the test-generation tasks.*

3) ChatGPT's PoV Demonstration Given Divergent Types of Prompts: Removing any element from P worsened ChatGPT's effectiveness. Among the variants, P_4 caused ChatGPT to work worst, producing zero successful PoV demonstration. This phenomenon implies that exemplar tests offer (1) important program structures for potential security tests, and (2) essential hints on vulnerability-triggering inputs. Without Lib tests, ChatGPT generated 11 tests throwing no error/exception, 11 tests wrongly mocking program entities, and 8 tests triggering irrelevant errors/exceptions. Although slightly better than P_4, P_3 also worsened ChatGPT significantly and only one security test was produced successfully. This may be because the removed Java class C holds lots of context, whose absence caused ChatGPT to create tests in a context-agnostic way, making the created tests irrelevant or invalid.

P_1, P_2, and P_5 had very similar effects on ChatGPT, as the tool produced 15, 14, and 14 security tests given the prompts derived from each of them. All these numbers are much lower than the number reported for the default template P: 24. This implies that the elements removed by individual templates (iii, iv, vii) provide valuable signals to ChatGPT, to help it identify and focus on the vulnerable APIs, function-to-test, and specialized vulnerability. While (v) and (vi) provide as much relevant code as possible for ChatGPT to refer to, the other elements (iii, iv, vii) guide ChatGPT to pay special attention to the most important content in the relevant code.

TABLE V: The input information required by the default setting of different tools (Yes: ✓, No: ✗)

Information	SIEGE	TRANSFER	ChatGPT
Vulnerable API	✓	✓	✓
M	✗	✓	✓
C	✗	✓	✓
Exemplar test	✗	✓	✓
Vulnerability ID	✗	✗	✓
All .class files of App	✓	✓	✗
JAR file of Lib	✓	✓	✗
Vulnerable line number in Lib	✓	✗	✗

Finding 4: *Among the five information elements covered by the default prompt template P, all elements played an important role to help ChatGPT effectively generate security tests. In particular, (v) and (vi) were more important than (iii), (iv), and (vii).*

D. Tool Comparison (RQ3)

Two tools were recently proposed to automatically generate security tests: SIEGE [64] and TRANSFER [67]. SIEGE adopts a genetic algorithm (GA). For any $\langle App, Lib \rangle$ pair, SIEGE must be executed under the project folder of App, which includes all compiled .class files of App and JAR files of library dependencies (including Lib). As shown in Table V, SIEGE also requires users to specify the search target (i.e., the coverage goal for tests-to-generate), including the vulnerable API and vulnerable line number in Lib. SIEGE reuses EvoSuite [58]—the popularly used test generation tool—to generate tests, select tests based on their closeness to the specified target, and evolve those tests with some randomness to get better tests. SIEGE stops when the time budget is used up or some tests perfectly match the target.

Similar to SIEGE, TRANSFER also generates security tests using GA. However, as shown in Table V, TRANSFER requires users to provide slightly different inputs: instead of including the vulnerable line number, users should designate C, M, and an exemplar test from Lib. It conducts program static analysis to create both call graphs and control-flow graphs for App, and uses dynamic instrumentation to assess test coverage. It executes and dynamically instruments the exemplar test, to identify program states relevant to the vulnerability, and to extract conditions that must be satisfied by any generated security test. Finally, TRANSFER adopts the extracted information to guide EvoSuite and derive security tests. Due to the usage of exemplar test and advanced program analysis techniques, TRANSFER manifested better effectiveness than SIEGE [67].

1) Experiment: We prepared the inputs required by SIEGE and TRANSFER for 49 $\langle App, Lib \rangle$ pairs, and executed both tools. Note that the two tools have full access to the .class files of each App and the JAR file of each Lib, while ChatGPT can only access the partial code described in prompts. For accurate comparison, we properly prepared the inputs for individual tools. We copy-and-pasted each tool-generated test to appropriate places. We leveraged the build process to reveal compilation errors. We also applied minor fixes to obvious and simple compilation errors. If compilation succeeded, we executed App with the generated test to observe runtime behaviors. If any exception or runtime error was thrown, we studied the exception/error message, inspected intermediate program states via step-by-step debugging, and discussed the relevance among authors until reaching a consensus.

TABLE VI: The comparison between ChatGPT and state-of-the-art tools on our dataset

	Tool Applicability (A)	Test Compilability (C)	PoV Demonstration (D)
ChatGPT	49	38 (26 + 12 *)	24 (19 + 5*)
ChatGPT w/o vulnerability ID (i.e., P_5)	49	36 (18 + 18*)	14 (7 + 7*)
TRANSFER	16	13 (9 + 4*)	4 (3 + 1*)
SIEGE	1	1	0

* marks the number of tests that compile after we applied minor fixes

2) Results: As shown in Table VI, ChatGPT outperformed current tools considerably by having much better tool applicability, test compilability, and PoV demonstration. It generated tests for all 49 $\langle App, Lib \rangle$ pairs, while TRANSFER only generated test functions or code snippets for 16 pairs. SIEGE worked much worse, creating a test for only one pair.

In terms of compilability, 13 out of the 16 tests generated by TRANSFER are compilable, including 4 tests with minor fixes applied; 3 of the 16 tests fail compilation due to their usage of unknown entities. For those 16 tasks, ChatGPT generated 15 compilable tests, including 3 tests fixed with minor edits. The only test output by SIEGE compiles successfully. For that same task, ChatGPT also generated a compilable test. These phenomena imply that ChatGPT outperforms existing tools by generating more compilable or easy-to-compile tests. There is no task showing either existing tool to outperform ChatGPT.

In terms of PoV demonstration, only four of the tests output by TRANSFER trigger vulnerabilities; For the four Apps handled well by TRANSFER, ChatGPT also generates tests that trigger the same vulnerabilities, indicating that it produces a superset of the effective PoVs found by TRANSFER. Among the remaining nine compilable tests produced by TRANSFER, six tests execute smoothly, without any error or exception; three tests trigger irrelevant exceptions. The only test by SIEGE fails to trigger any vulnerability, because it throws an irrelevant exception. These observations indicate that ChatGPT not only outperforms TRANSFER but also discovered more diverse and effective vulnerability-triggering PoV test

When ChatGPT and TRANSFER perform differently, one may wonder whether (1) the extra input of vulnerability ID that ChatGPT takes or (2) the distinct tools' working mechanisms explain the observed differences. To characterize the contributions of both factors between ChatGPT and TRANSFER, in Table VI, we also included a variant approach of the default ChatGPT usage by using template P_5. As shown in the table, when ChatGPT took in part of its inputs that are also required by TRANSFER, it achieved better tool applicability (49 vs. 16), test compatibility (36 vs. 13), and PoV demonstration (14 vs. 4) than TRANSFER. Meanwhile, the variant approach worked less effectively than our default ChatGPT usage. These results imply that between ChatGPT and TRANSFER, both (1) the vulnerability ID that ChatGPT takes and (2) the distinct tools' working mechanisms contribute to the observed

TABLE VII: The comparison between ChatGPT and TRANSFER on the TRANSFER's dataset that we restored

	Tool Applicability (A)	Test Compilability (C)	PoV Demonstration (D)
ChatGPT	28	20 (9 + 11*)	16 (9 + 7*)
ChatGPT w/o vulnerability ID (i.e., P_5)	28	24 (13+11*)	14 (10+4*)
TRANSFER	12	9 (3 + 6*)	5 (3 + 2*)

differences, although (2) plays a more important role.

We further inspected the cases where TRANSFER worked worse than ChatGPT, and summarized three major limitations of the tool design. First, TRANSFER adopts EvoSuite, to generate tests and execute the method-to-test M. However, EvoSuite is not quite effective; some or even most of the tests generated by EvoSuite do not execute M at all. Second, TRANSFER has difficulty synthesizing or mocking complex input parameters for M. For instance, it could not synthesize a parameter of type `net.sourceforge.pmd.RuleSet`, but ChatGPT mocked such an object via the Mockito framework.

Third, TRANSFER has difficulty incorporating the knowledge embedded in exemplar tests into test generation. For instance, CODEC-134 is related to two vulnerable APIs: `Base64.decode(...)` and `Base64.decodeBase64(...)`. We provided both TRANSFER and ChatGPT inputs relevant to that vulnerability, including the Lib test to show PoV of one API `Base64.decode(...)`, and a client Java class to call the other API `Base64.decodeBase64(...)`. We used both tools to generate a security test for the client code. Unfortunately, TRANSFER could not reuse any domain knowledge from the Lib test, but ChatGPT successfully achieved that.

SIEGE worked much worse than ChatGPT. The major reason is that SIEGE has very limited applicability, probably due to implementation issues. Among the 49 $\langle App, Lib\rangle$ pairs, SIEGE was only applicable to a single pair.

Finding 5: *ChatGPT outperformed both TRANSFER and SIEGE on our dataset. No security test was successfully generated by TRANSFER or SIEGE, but not by ChatGPT. ChatGPT has great potentials in generating security tests.*

3) Additional Experiments and Results: For fair comparison, we also tried to apply tools to the datasets mentioned by papers of SIEGE and TRANSFER [64], [67]. Unfortunately, as we mentioned before, the open-sourced dataset of TRANSFER [48] has 21 apps that are not quite usable in our evaluation. Seven of the Apps lack essential details for successful build or execution; six Apps depend on secure instead of vulnerable versions of Libs; five Apps have no vulnerable API calls or Lib test; two Apps wrap vulnerable API calls with security sanity checks to eliminate potential exploits; one App calls the vulnerable API in the test file. To fully leverage TRANSFER's dataset in our tool-comparison experiment, we manually replaced the security dependencies with vulnerable ones in six Apps, and still used the single App that calls vulnerable API in the test file.

In this way, we experimented with 28 of the 42 $\langle App, Lib\rangle$ pairs in TRANSFER's dataset. As shown in Table VII, by applying ChatGPT and TRANSFER to that dataset, we found ChatGPT outperform TRANSFER in all metrics. We also applied ChatGPT's variant approach based on P_5, as this variant takes only inputs that the default approach commonly shares with TRANSFER. This variant also outperformed TRANSFER in all metrics, indicating the stronger power of ChatGPT in generating security tests.

Finding 6: *ChatGPT outperformed TRANSFER on the TRANSFER's dataset that we restored, no matter whether ChatGPT takes in solely the inputs shared with TRANSFER or those together with the unique input vulnerability ID.*

TABLE VIII: The comparison between ChatGPT and SIEGE on SIEGE's dataset

	Tool Applicability (A)	Test Compilability (C)	PoV Demonstration (D)
ChatGPT w/o exemplar test (i.e., P_4)	11	10	0
ChatGPT taking in only vulnerable API and C	11	7	0
SIEGE	9	9	0

Although SIEGE's dataset is publicly available, the dataset includes no Lib test for ChatGPT to refer to. Therefore, when applying ChatGPT to SIEGE's dataset, by default, our prompts do not include any Lib test. Furthermore, we also explored a variant usage of ChatGPT by specifying only the vulnerable API and client code C, so that this variant takes in no more input than SIEGE.

As shown in Table VIII, SIEGE generated tests for 9 of the 11 tasks; the tests are compilable, but no test shows PoV. In comparison, ChatGPT-without-exemplar-test generated 11 tests; 10 of the tests are compilable, and 1 test is uncompilable due to wrong type casting. None of the tests show PoV. This is as expected. As mentioned in Section IV-C, when no exemplar test is offered, ChatGPT cannot effectively generate security tests. ChatGPT's variant usage that takes no more input than SIEGE also generated 11 tests; 7 of the tests are compiable and 0 test shows PoV. The result implies that compared with SIEGE, ChatGPT always has a better tool applicability, although the test compilability and PoV demonstration of ChatGPT-generated tests are not necessarily better.

Finding 7: *ChatGPT is always more applicable than SIEGE. However, when no exemplar test is provided, ChatGPT does not achieve better PoV demonstration.*

E. Comparison between Zero-shot Prompting and Few-shot Prompting (RQ4)

In all experiments we have explained so far, we adopted the zero-shot prompting technique. Namely, we did not provide any exemplar question-answer pair to ChatGPT, to enable in-context learning where demonstrations are provided in prompts to steer the model for better performance. One hypothesis we had is that ChatGPT might work better given few-shot prompts. To validate that hypothesis, we conducted an experiment by sending ChatGPT question-answer pair(s) in each prompt. As shown in Fig. 4, a three-shot prompt first describes

An Exemplar Prompt for Few-Shot Learning

```
Assume you're a security expert. I provide the prompt and you
provide the corresponding response. I will provide the examples with
concrete values for the place holders [] in the following prompt
template, and the result you're supposed to produce:

Prompt template:
Write a JUnit test for function: [function-name] in "CLIENT CODE".
The test should verify whether the function [function-name] is affected
by the vulnerability: [vulnerability-ID], or affected by vulnerable APIs:
[vulnerable-API-list]. Mimic the test function "TEST FUNCTION:"
[test-function-name] I provide to generate the input in your test. For
the provided code "CLIENT CODE", mock the classes and functions
that are not Java SE APIs and have no definition provided.

Example 1:
[vulnerability-ID]: CODEC-134
[vulnerable-API-list]: Base64.decode, Base64.decodeBase64
[function-name]: decode
TEST Function: ...
CLIENT CODE: ...
Response you provide should be: ...

Example 2:
[vulnerability-ID]: CVE-2018-12418
[vulnerable-API-list]: Archive.Archive
[function-name]: createInstance
TEST Function: ...
CLIENT CODE: ...
Response you provide should be: ...

Example 3:
[vulnerability-ID]: CVE-2020-13956
[vulnerable-API-list]: CloseableHttpClient.execute,
URIUtils.extractHost
[function-name]: post
TEST Function: ...
CLIENT CODE: ...
Response you provide should be: ...

-----------------
[vulnerability-ID]: HTTPCLIENT-1803
[vulnerable-API-list]: URIBuilder.URIBuider,
URIBuilder.setHost,URIBuilder.build, URIBuilder.toString
[function-name]: startElement
```

Fig. 4: A three-shot prompt that provides examples for three test-generation tasks

the default template P we used for zero-shot learning. It then provides three pairs of (1) input parameters to customize P, and (2) exemplar output we manually crafted in response to those customized test-generation tasks. Finally, it offers a fourth set of input parameters to customize P, making ChatGPT generate a test accordingly.

To improve the representativeness of our evaluation, we experimented with two types of few-shot prompts: one-shot and three-shot prompts. To avoid the randomness contributed by diverse question-answer pairs, for each type of prompts, we intentionally provided the same set of question-answer pair(s). Namely, among the 49 $\langle App, Lib \rangle$ pairs in our dataset, for 3-shot prompts, we randomly picked 3 pairs for example construction, and used the remaining 46 pairs for evaluation. For one-shot prompts, we used one of the three pairs for example construction, and reused the evaluation set for three-shot prompts mentioned above. By splitting the dataset in this way, we ensure that (1) there is no overlap between the exemplar question-answer pairs and evaluation questions, and (2) different prompting techniques are evaluated with exactly the same data portion.

TABLE IX: The comparison between zero-shot and few-shot prompting of ChatGPT on the dataset of 46 $\langle App, Lib \rangle$ pairs

	Tool Applicability (A)	Test Compilability (C)	PoV Demonstration (D)
ChatGPT with zero-shot learning (default)	46	36 (24 + 12*)	22 (17 + 5*)
ChatGPT with one-shot learning	46	32 (22 + 10*)	14 (11 + 3*)
ChatGPT with three-shot learning	46	37 (30 + 7*)	24 (23 + 1*)

Table IX compares the results of few-shot prompts against those of our default prompts. As shown in the table, with one-shot prompts, ChatGPT has worse test compilability (32 vs. 36), and worse PoV demonstration (14 vs. 22). However, with three-shot prompts, ChatGPT has slightly better test compilability (37 vs. 36) and better PoV demonstration (24 vs. 22). Surprisingly, including question-answer pairs into prompts does not guarantee improvements, but may worsen the results. Two possible reasons can explain the observed phenomena. First, when one-shot prompts are provided, the only question-answer pair mentioned is insufficient to steer ChatGPT for better performance, but may confuse the model to output less optimal results. Second, when three-shot prompts are offered, the multiple question-answer pairs mentioned present more demonstrations to better enable in-context learning.

Finding 8: *Compared with zero-shot prompting, one-shot prompting makes ChatGPT perform worse, while three-shot prompts improves ChatGPT's performance. Namely, few-shot prompting does not necessarily help improve the quality of generated test.*

F. Comparison across the different LLMs with default prompt settings (RQ5)

To assess the consistency of PoV test generation for client projects among different LLMs, we evaluated five state-of-the-art models : three closed-source LLMs (GPT-4.0, Gemini-2.5-pro-preview, and Claude-3.7-sonnet) and two open-source LLMs (Llama3.3-70b and Deepseek-chat-v3). All experiments were conducted through API calls with the temperature set to 0 to minimize randomness and ensure consistent outputs.

Table X compares each model's performance based on three metrics. As shown in the table, the results are consistent with what we observed in the ChatGPT experiment. They demonstrate better performance than TRANSFER and SIEGE in test generation across the 49 client applications in our dataset. Claude-3.7-sonnet achieved the highest initial test compilability (29), Gemini-2.5-pro-preview generated 25 initially compilable tests. The open-source models Llama3.3-70b and Deepseek-chat-v3 generated fewer initially compilable test cases, which are 20 and 19 respectively. After applying minor fixes, all models generated more than 30 compilable tests.

TABLE X: The comparison among different LLMs with default prompt settings on the dataset of 49 $\langle App, Lib \rangle$ pairs

	Tool Applicability (A)	Test Compilability (C)	PoV Demonstration (D)
GPT-4.0	49	35 (20 + 15*)	23 (16 + 7*)
Gemini-2.5-pro-preview	49	34 (25 + 9*)	18 (13 + 5*)
Claude-3.7-sonnet	49	35 (29 + 6*)	17 (14 + 3*)
Llama3.3-70b	49	36 (20 + 16*)	23 (13 + 10*)
Deepseek-chat-v3	49	32 (19 + 13*)	20 (13 + 7*)
ChatGPT (ChatBox with GPT-4.0)	49	38 (26 + 12*)	24 (19 + 5*)

For PoV demonstration (D), Llama3.3-70b, an open-source LLM, generated a comparable number of PoV demonstrations (13 + 10*) to GPT-4.0. Among these two models, Llama3.3-70b generated two unique tests that ChatGPT did not output. GPT-4.0 with temperature 0 performed consistently with ChatGPT, across all projects except one. The marginal difference was likely due to variations in system prompts and hyperparameter configuration, between the chat interface and API call. Deepseek-chat-v3, Gemini-2.5-pro-preview and Claude-3.7-sonnet generated fewer PoV demonstrations. Importantly, despite ChatGPT's better overall performance, other models successfully demonstrated PoVs for vulnerabilities that ChatGPT missed.

We found that 12 projects had the PoV uniquely generated by only a single model other than ChatGPT. Specifically, 5 of the 12 projects were uniquely and successfully handled by Claude-3.7-sonnet, 2 projects by GPT-4.0, 3 by Gemini-2.5-pro-preview, and 2 by Llama3.3-70b. This indicates that different LLMs can uncover vulnerabilities in unique scenarios. Meanwhile, 26 of the vulnerable projects had successful PoVs generated by multiple models, suggesting overlap in their PoV demonstration generation capabilities. Of these, seven projects had successful PoVs generated by all five evaluated LLMs. However, for 11 projects, no LLMs could generate effective PoV tests, primarily for two reasons: (1) Mockito-related errors hinder the generation of compilable tests; (2) the generated tests, although compilable, fail to trigger and reveal the target vulnerability.

Our comparative analysis reveals that closed-source models generally demonstrated higher initial Test compilability than their open-source models. There is a substantial overlap in the PoV generation capabilities across different LLMs. LLMs encounter common challenges when getting applied to 11 of the client applications. These challenges are concerning handling complex code context (e.g., failing to Mock the objects and functions), and crafting tests that effectively trigger specific vulnerabilities.

Finding 9: *Compared to ChatGPT, GPT demonstrates similar performance in both Test Compilability and PoV Demonstration. There is a substantial overlap in the PoV generation capabilities across different LLMs; LLMs also encounter common challenges when getting applied to 11 of the client applications.*

V. Threats to Validity

Threats to External Validity: Our observations may be limited to the experiment datasets. Among the 49 Apps in our dataset, 9 functions-under-test call vulnerable APIs indirectly, and 26 functions-under-test have parameter lists different from those of the vulnerable APIs they call. The inclusion of these non-trivial cases helps ensure the representativeness of our observations. In the future, to make our findings more generalizable, we will explore more vulnerabilities, and experiment with more LLMs as well as programs written in other languages.

Our current investigation relies on exemplar security tests from Libs to show PoV. If a Lib does not have such tests, our empirical findings may not generalize well to Apps using that Lib. Prior work [67] shows that most Lib vulnerabilities are fixed with test cases, implying the wide existence of exemplar tests and good generalizability of our findings. Future work can overcome this limitation by mining reusable proof-of-concept malicious inputs online. Because some vulnerabilities share malicious inputs (e.g., invalid inputs to realize directory traversal), if we reuse inputs across the vulnerabilities under the same category, we do not necessarily need any exemplar test for a particular Lib.

Threats to Internal Validity: We experimented with the default setting of ChatGPT, without controlling or tuning any parameter it defines. LLMs exhibit inherent non-deterministic behavior [57], [81], [86], causing identical prompts to produce varying outputs. This stochasticity threatens experimental validity. To quantify this effect, we submitted 25 test-generation tasks five times each to ChatGPT-4.0 across 125 independent conversation sessions, yielding consistent outputs in 113 trials (90% consistency). Additionally, we validated these findings through experiments with five LLMs, setting the temperature parameter to 0 to minimize stochastic behavior. The reproducible results across controlled conditions confirm our initial findings with ChatGPT. Based on these results, we believe that the internal randomness of ChatGPT did not significantly compromise the validity of our experimental outcomes.

Our manual verification during API invocation validation and result validation poses a potential threat to internal validity, as the verification steps require human judgment to determine whether generated PoV tests trigger the same vulnerability as the target library. To address this limitation, we developed explicit criteria for vulnerability classification and employed multiple researchers to independently verify a subset of our findings, ensuring consistency in our evaluation methodology.

Threats to Construct Validity: ChatGPT was trained on large collections of text data available online (e.g., books, articles, and web pages). Thus, it was likely trained with existing vulnerability entries (CVEs and issue reports), software libraries, and apps, but was not trained with app-specific security tests. This is because app-specific security tests are rare on the Internet. None of the tests we generated ever existed for those apps. ChatGPT would not know about these app-specific security tests beforehand, so our experiment does not suffer from the data leakage issue of machine learning.

VI. Related Work

The related work includes automatic vulnerability repair, security test generation, and LLM-based research.

A. Automatic Vulnerability Repair (AVR)

Various approaches were proposed to generate repair patches, to potentially accelerate manual security analysis and vulnerability removal [54], [55], [59], [72]–[74], [77], [102], [104], [106]. For instance, VuRLE [74] and SEADER [106] learned vulnerability-repair patterns from $\langle insecure, secure \rangle$ code examples; they both used the patterns to detect vulnerabilities and suggest repairs. Search-based program repair tools [72], [73], [77], [102] integrate frequently-used repair patterns or widespread code templates; given a buggy program, they navigate the search space to generate candidate repairs and validate each repair via testing. Learning-based vulnerability repair tools [54], [55], [59], [104] mainly leverage machine-learning models pre-trained on labeled or unlabeled code corpus, to derive generic language representation or infer correlation between buggy code and program repair. The tools then apply transfer learning to fine-tune those models for security vulnerability repair with a limited labeled corpus.

Our research does not repair security vulnerabilities, but the tests it intends to generate are closely related to AVR. Namely, when security tests are successfully generated, they can be used by AVR to decide whether a candidate repair eliminates any vulnerability.

B. Security Test Generation

Tools were built to generate security tests [9], [32], [49], [50], [52], [53], [56], [60], [62], [75], [80], [94], [100]. Specifically, Marback et al. [75] and Xu et al. [100] created tools, to partially automate the procedure of generating security tests from threat models (e.g., threat trees or nets). Namely, these approaches first reveal potential attack paths by automatically traversing hand-crafted threat models, and then convert paths to test cases via tool automation or manual effort. However, users may have insufficient domain knowledge to manually model all threats/attacks, or to accurately convert attack paths to tests. Consequently, these approaches are ineffective in practice for test creation.

Traditional verification takes in a program and a specification of safety, and verifies whether the program satisfies the safety specification. Automatic exploit generation (AEG) [50], [52], [53], [60] twists program verification, by replacing the safety property with an exploitability property, and the verification process becomes finding a program path where the exploitability property holds. For instance, Ganapathy et al. [60] explore API-level exploitability with bounded model checking (BMC). AEG often suffers from scalability challenges (e.g., path explosion and the NP-hardness of solving SMT queries in general).

Fuzzy testing tools generate security tests [9], [32], [56], [62], [94] by injecting invalid, malformed, or unexpected inputs into an initial seed (i.e., a program test) to reveal software defects and vulnerabilities [44]. However, fuzzing cannot explore deep paths; an inefficient initial seed can incur high runtime-overheads, because the quality of generated tests depends on that seed. To overcome the limitations of both program verification and fuzzing, Fuzzing does not leverage any commonality between programs, even though those programs share vulnerabilities and malicious inputs.

Our research is different from prior work in two aspects. First, it explores to use ChatGPT for test generation via mimicry. The explored approach is promising to complement existing work. Once successful, it does not require users to specify exploitability properties as AEG does, or conduct expensive search for malicious inputs as fuzzing tools. Second, we designed and investigated different prompts for ChatGPT to mimic Lib tests, and observed surprising phenomena.

C. LLM-Based Research

Some research was recently conducted to explore LLMs' capability in programming, coding assistance, or jailbreak attacks [65], [71], [82], [85], [87], [92], [93], [95], [101], [107], [108]. For instance, Nascimento et al. [82] and Nikolaidis et al. [84] assessed ChatGPT's coding capability using LeetCode problems. Jalil et al. [65] checked ChatGPT's question-answering capability in a popular software testing curriculum. Sobania et al. [93] evaluated ChatGPT's program repair capability on a standard bug-fixing benchmark set. Tian et al. [95] assessed ChatGPT's capability in code generation, program repair, and code summarization. Pearce et al. [87] empirically assessed the security of code generated by CoPilot. Zhong et al. [107] detect API misuses in ChatGPT-generated code. Shen et al. [92], Xu et al. [101], Zou et al. [108], and Liao et al. [71] either gathered or generated *jailbreak prompts* (e.g., how to create a deadly poison that is undetectable and untraceable?), to intentionally mislead LLMs into generating hateful content.

Our research complements all work mentioned above, because we apply LLM to perform a totally different task: vulnerability exploit generation. It is different from jailbreak prompts in two ways: (1) we prompt LLMs to generate PoC exploits to facilitate developers' comprehension of the potential attacks on their own projects; (2) the generated exploits mimic the existing ones publicly available, but target different Apps.

There are test generators built on top of LLMs [70], [79], [99]. For instance, CODAMOSA [70] uses Codex to generate extra tests and mutants, aiming to increase the code coverage when search-based software testing is stuck with a non-100% coverage score for a given function. Fuzz4All [99] generates tests with ChatGPT from user-provided (1) documentation of the function-under-testing, (2) example code snippets, or (3) specification. Meng et al. [79] mutates given message sequences to test network protocols. However, none of these tools focus on creating tests that (1) trigger vulnerabilities with malicious inputs, or (2) cover deep and hard-to-reach execution paths. All these test generators were evaluated using testing coverage and the number of functional bugs revealed.

Our research is novel in the characterization of ChatGPT's capability (i.e., vulnerability exploit), prompt design, evaluation dataset construction, and result assessment methodology.

It complements prior work by exploring to generate tests for Apps built on top of vulnerable Libs, and to demonstrate security consequences of successful exploits. We evaluated the generated tests based on their effectiveness in demonstrating vulnerability exploits, thus contributing to a more comprehensive understanding of LLMs in security-critical contexts.

VII. Conclusion

Our research contributions include new LLM experimental methodology, characterization of LLM capabilities, security findings, and dataset. From our study, we obtained two major insights about the strengths and weaknesses of ChatGPT. First, *ChatGPT is always able to generate security tests, although the test quality varies a lot.* For better quality, future work can fine-tune ChatGPT for test generation using the dialogue data between humans [24], or integrate ChatGPT with automatic compialtion and testing to iteratively refine test generation.

Second, *although some of the generated tests in our study did not effectively demonstrate PoV for known vulnerabilities, they surprisingly revealed new vulnerabilities.* This implies that ChatGPT is also promising in generating tests to reveal new software bugs or vulnerabilities. Future work can integrate ChatGPT with existing test generation tools, to better generate tests and reveal new vulnerabilities or bugs more efficiently.

Our current investigation adopts ChatGPT as a human assistant, as we manually gathered vulnerability-related information (i.e., the elements (ii)–(vii) mentioned in Section III-B), and provided that information to ChatGPT. It means that with our approach, users of ChatGPT need to contribute some manual effort before getting successfully generated tests. In the future, we will further reduce such manual effort, by creating more advanced mining techniques to crawl code bases and vulnerability databases for relevant information.

References

[1] GitHub - nearform / gammaray: Node.js vulnerability scanner. https://github.com/nearform/gammaray, 2019.
[2] OWASP Dependency-Check. https://owasp.org/www-project-dependency-check/, 2020.
[3] Snyk vulnerability database. http://snyk.io/vuln, 2020.
[4] Supply chain attacks on open source software grew 650% in 2021. https://techmonitor.ai/technology/cybersecurity/supply-chain-attacks-open-source-software-grew-650-percent-2021, 2021.
[5] Supply chain attacks show why you should be wary of third-party providers. https://www.csoonline.com/article/3191947/supply-chain-attacks-show-why-you-should-be-wary-of-third-party-providers.html, 2021.
[6] Log4Shell a year on. https://usa.kaspersky.com/blog/log4shell-still-active-2022/27531/, 2022.
[7] About Dependabot alerts. https://docs.github.com/en/code-security/dependabot/dependabot-alerts/about-dependabot-alerts, 2023.
[8] alibaba / fastjson. https://github.com/alibaba/fastjson, 2023.
[9] american fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2023.
[10] apache / commons-io. https://github.com/apache/commons-io, 2023.
[11] apache / cxf. https://github.com/apache/cxf, 2023.
[12] apache / httpcomponents-client. https://github.com/apache/httpcomponents-client, 2023.
[13] Apache Tika. https://tika.apache.org, 2023.
[14] Automatic fixing with snyk fix - Snyk User Docs. https://docs.snyk.io/snyk-cli/test-for-vulnerabilities/automatic-remediation-with-snyk-fix, 2023.
[15] Codec. https://commons.apache.org/proper/commons-codec/, 2023.
[16] Commons Compress - Overview. https://commons.apache.org/proper/commons-compress/, 2023.
[17] CVE-2020-28052 Detail. https://nvd.nist.gov/vuln/detail/cve-2020-28052, 2023.
[18] Dom4j. https://dom4j.github.io, 2023.
[19] FasterXML / jackson-databind. https://github.com/FasterXML/jackson-databind, 2023.
[20] FasterXML / jackson-dataformats-binary. https://github.com/FasterXML/jackson-dataformats-binary, 2023.
[21] FasterXML / jackson-modules-java8. https://github.com/FasterXML/jackson-modules-java8, 2023.
[22] GitHub security advisories. https://github.com/advisories, 2023. Accessed on June 12, 2023.
[23] haraldk / TwelveMonkeys. https://github.com/haraldk/TwelveMonkeys, 2023.
[24] How does ChatGPT actually work? https://www.zdnet.com/article/how-does-chatgpt-work/, 2023.
[25] Inside ChatGPT's Brain: Large Language Models. https://serokell.io/blog/language-models-behind-chatgpt, 2023.
[26] junrar / junrar. https://github.com/junrar/junrar, 2023.
[27] Mockito. https://site.mockito.org/, 2023. Accessed on June 12, 2023.
[28] netplex / json-smart-v1. https://github.com/netplex/json-smart-v1, 2023.
[29] netplex / json-smart-v2. https://github.com/netplex/json-smart-v2, 2023.
[30] npm-audit. https://docs.npmjs.com/cli/v9/commands/npm-audit, 2023.
[31] OpenRefine. https://github.com/OpenRefine/OpenRefine, 2023.
[32] OSS-Fuzz. https://google.github.io/oss-fuzz/, 2023.
[33] OWASP / json-sanitizer. https://github.com/OWASP/json-sanitizer, 2023.
[34] OWASP Top Ten. https://owasp.org/www-project-top-ten/, 2023.
[35] Plexus Archiver Component. https://codehaus-plexus.github.io/plexus-archiver/index.html, 2023.
[36] Retire.js. https://retirejs.github.io/retire.js/, 2023.
[37] sonatype-nexus-community / auditjs: Audits an NPM package.json file to identify known vulnerabilities. https://github.com/sonatype-nexus-community/auditjs, 2023.
[38] spring-projects / spring-data-commons. https://github.com/spring-projects/spring-data-commons, 2023.
[39] spring-projects / spring-security. https://github.com/spring-projects/spring-security, 2023.
[40] srikanth-lingala / zip4j. https://github.com/srikanth-lingala/zip4j, 2023.
[41] stleary / JSON-java. https://github.com/stleary/JSON-java, 2023.
[42] Test - Snyk User Docs. https://docs.snyk.io/snyk-cli/commands/test, 2023.
[43] The Legion of the Bouncy Castle. https://www.bouncycastle.org, 2023.
[44] What Is Fuzz Testing and How Does It Work? — Synopsys. https://www.synopsys.com/glossary/what-is-fuzz-testing.html, 2023.
[45] xerial / snappy-java. https://github.com/xerial/snappy-java, 2023.
[46] XStream. https://x-stream.github.io, 2023.
[47] ZT Zip. https://github.com/zeroturnaround/zt-zip, 2023.
[48] soarsmu/transfer. https://github.com/soarsmu/transfer, 2024.
[49] K. M. Alshmrany, M. Aldughaim, A. Bhayat, and L. C. Cordeiro. Fusebmc: An energy-efficient test generator for finding security vulnerabilities in c programs. In F. Loulergue and F. Wotawa, editors, *Tests and Proofs*, pages 85–105, Cham, 2021. Springer International Publishing.
[50] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.
[51] H. Booth, D. Rike, and G. A. Witte. The national vulnerability database (nvd): Overview. 2013.
[52] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 143–157, 2008.
[53] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, 2012.
[54] Z. Chen, S. Kommrusch, and M. Monperrus. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 49(1):147–165, 2023.
[55] J. Chi, Y. Qu, T. Liu, Q. Zheng, and H. Yin. Seqtrans: Automatic vulnerability fix via sequence to sequence learning. *IEEE Transactions on Software Engineering*, 49(2):564–585, 2023.
[56] J. Demott, D. Richard, R. Enbody, D. William, and W. Punch. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. In *Black Hat and DEFCON*, 07 2007.

[57] J. Dietrich and A. Hollstein. Performance and reproducibility of large language models in named entity recognition: Considerations for the use in controlled environments. *Drug Safety*, 48(3):287–303, 2025.

[58] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[59] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung. Vulrepair: a t5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 935–947, New York, NY, USA, 2022. Association for Computing Machinery.

[60] V. Ganapathy, S. A. Seshia, S. Jha, T. W. Reps, and R. E. Bryant. Automatic discovery of api-level exploits. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 312–321, New York, NY, USA, 2005. Association for Computing Machinery.

[61] K. Garrett, G. Ferreira, L. Jia, J. Sunshine, and C. Kästner. Detecting suspicious package updates. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 13–16. IEEE, 2019.

[62] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft. *Queue*, 10(1):20–27, jan 2012.

[63] R. Hat. Red hat. *Seam-Contextual Components. A Framework for Java EE*, 5, 2007.

[64] E. Iannone, D. Di Nucci, A. Sabetta, and A. De Lucia. Toward automated exploit generation for known vulnerabilities in open-source libraries. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 396–400. IEEE, 2021.

[65] S. Jalil, S. Rafi, T. D. LaToza, K. Moran, and W. Lam. Chatgpt and software testing education: Promises & perils. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 4130–4137, Los Alamitos, CA, USA, apr 2023. IEEE Computer Society.

[66] M. M. A. Kabir, Y. Wang, D. Yao, and N. Meng. How do developers follow security-relevant best practices when using npm packages? In *2022 IEEE Secure Development Conference (SecDev)*, pages 77–83, Los Alamitos, CA, USA, oct 2022. IEEE Computer Society.

[67] H. J. Kang, T. G. Nguyen, B. Le, C. S. Păsăreanu, and D. Lo. Test mimicry to assess the exploitability of library vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 276–288, 2022.

[68] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, et al. Cognicrypt: supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 931–936. IEEE, 2017.

[69] P. Ladisa, H. Plate, M. Martinez, O. Barais, and S. E. Ponta. Towards the detection of malicious java packages. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 63–72, 2022.

[70] 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 International Conference on Software Engineering*, ICSE '23, pages 919–931. IEEE Press, 2023.

[71] Z. Liao and H. Sun. Amplegcg: Learning a universal and transferable generative model of adversarial suffixes for jailbreaking both open and closed llms, 2024.

[72] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. Tbar: revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 31–42, New York, NY, USA, 2019. Association for Computing Machinery.

[73] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 166–178, New York, NY, USA, 2015. Association for Computing Machinery.

[74] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng. Vurle: Automatic vulnerability detection and repair by learning from examples. In S. N. Foley, D. Gollmann, and E. Snekkenes, editors, *Computer Security – ESORICS 2017*, pages 229–246, Cham, 2017. Springer International Publishing.

[75] A. Marback, H. Do, K. He, S. Kondamarri, and D. Xu. Security test generation using threat trees. In *2009 ICSE Workshop on Automation of Software Test*, pages 62–69, 2009.

[76] B. Martin, M. Brown, A. Paller, D. Kirby, and S. Christey. Cwe. *SANS top*, 25, 2011.

[77] M. Martinez and M. Monperrus. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In T. E. Colanzi and P. McMinn, editors, *Search-Based Software Engineering*, pages 65–86, Cham, 2018. Springer International Publishing.

[78] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. A. Argoty. Secure coding practices in java: Challenges and vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering*, pages 372–383, 2018.

[79] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[80] R. Metta, R. K. Medicherla, and S. Chakraborty. Bmc+fuzz: Efficient and effective test generation. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1419–1424, 2022.

[81] A. Nafar, K. B. Venable, and P. Kordjamshidi. Probabilistic reasoning in generative large language models. *arXiv e-prints*, pages arXiv–2402, 2024.

[82] N. Nascimento, P. Alencar, and D. Cowan. Comparing software developers with chatgpt: An empirical investigation, 2023.

[83] D. C. Nguyen, E. Derr, M. Backes, and S. Bugiel. Up2dep: Android tool support to fix insecure code dependencies. In *Annual Computer Security Applications Conference*, ACSAC '20, pages 263–276, New York, NY, USA, 2020. Association for Computing Machinery.

[84] N. Nikolaidis, K. Flamos, D. Feitosa, A. Chatzigeorgiou, and A. Ampatzoglou. The End of an Era: Can Ai Subsume Software Developers? Evaluating Chatgpt and Copilot Capabilities Using Leetcode Problems. http://dx.doi.org/10.2139/ssrn.4422122.

[85] N. Nikolaidis, K. Flamos, D. Feitosa, A. Chatzigeorgiou, and A. Ampatzoglou. The end of an era: Can ai subsume software developers? evaluating chatgpt and copilot capabilities using leetcode problems. *Evaluating Chatgpt and Copilot Capabilities Using Leetcode Problems*, 2023.

[86] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang. An empirical study of the non-determinism of chatgpt in code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–28, 2025.

[87] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768. IEEE, 2022.

[88] S. E. Ponta, H. Plate, and A. Sabetta. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, 25(5):3175–3215, 2020.

[89] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, pages 383–387. IEEE Press, 2019.

[90] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. Yao. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2455–2472, 2019.

[91] K. Rahkema and D. Pfahl. Swiftdependencychecker: Detecting vulnerable dependencies declared through cocoapods, carthage and swift pm. In *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, MOBILESoft '22, pages 107–111, New York, NY, USA, 2022. Association for Computing Machinery.

[92] X. Shen, Z. Chen, M. Backes, Y. Shen, and Y. Zhang. "do anything now": Characterizing and evaluating in-the-wild jailbreak prompts on large language models, 2023.

[93] D. Sobania, M. Briesch, C. Hanna, and J. Petke. An analysis of the automatic bug fixing performance of chatgpt, 2023.

[94] A. Takanen, J. Demott, C. Miller, and A. Kettunen. *Fuzzing for Software Security Testing and Quality Assurance, Second Edition*. Artech House, 2017.

[95] H. Tian, W. Lu, T. O. Li, X. Tang, S.-C. Cheung, J. Klein, and T. F. Bissyandé. Is chatgpt the ultimate programming assistant – how far is it?, 2023.

[96] D. L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta. Towards using source code repositories to identify software supply chain attacks. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 2093–2095, 2020.

[97] C. Vulnerabilities. Common vulnerabilities and exposures, 2005.
[98] Y. Wu, Z. Yu, M. Wen, Q. Li, D. Zou, and H. Jin. Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1046–1058. IEEE, 2023.
[99] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.
[100] D. Xu, M. Tu, M. Sanford, L. Thomas, D. Woodraska, and W. Xu. Automated security test generation with formal threat models. *IEEE Transactions on Dependable and Secure Computing*, 9(4):526–540, 2012.
[101] X. Xu, K. Kong, N. Liu, L. Cui, D. Wang, J. Zhang, and M. Kankanhalli. An llm can fool itself: A prompt-based adversarial attack, 2023.
[102] Y. Yuan and W. Banzhaf. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming . *IEEE Transactions on Software Engineering*, 46(10):1040–1067, Oct. 2020.
[103] S. Zander, G. Armitage, and P. Branch. A survey of covert channels and countermeasures in computer network protocols. *IEEE Communications Surveys & Tutorials*, 9(3):44–57, 2007.
[104] Q. Zhang, C. Fang, B. Yu, W. Sun, T. Zhang, and Z. Chen. Pre-trained model-based automated software vulnerability repair: How far are we? *IEEE Transactions on Dependable and Secure Computing*, 21(4):2507–2525, 2024.
[105] Y. Zhang, M. M. A. Kabir, Y. Xiao, D. Yao, and N. Meng. Automatic detection of java cryptographic api misuses: Are we there yet? *IEEE Transactions on Software Engineering*, 49(1):288–303, 2022.
[106] Y. Zhang, Y. Xiao, M. M. A. Kabir, D. D. Yao, and N. Meng. Example-based vulnerability detection and repair in java code. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, ICPC '22, pages 190–201, New York, NY, USA, 2022. Association for Computing Machinery.
[107] L. Zhong and Z. Wang. Can llm replace stack overflow? a study on robustness and reliability of large language model code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 21841–21849, 2024.
[108] A. Zou, Z. Wang, N. Carlini, M. Nasr, J. Z. Kolter, and M. Fredrikson. Universal and transferable adversarial attacks on aligned language models, 2023.