Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

arXiv:2310.00710v2 [cs.CR] 3 Oct 2023

How well does LLM generate security tests?

YING ZHANG, Virginia Tech, USA
WENJIA SONG, Virginia Tech, USA
ZHENGJIE JI, Virginia Tech, USA
DANFENG (DAPHNE) YAO, Virginia Tech, USA
NA MENG, Virginia Tech, USA

attacks from vulnerabilities in libraries

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. People refer to such attacks as **supply chain attacks**, the documented number of which has increased 742% in 2022. People 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 ask for 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.

In this study, we used ChatGPT-4.0 to generate security tests, and to demonstrate how vulnerable library dependencies facilitate the supply chain attacks to given Apps. We explored various prompt styles/templates, and found that ChatGPT generated tests for all 55 Apps, demonstrating 24 attacks successfully. It outperformed two state-of-the-art security test generators—TRANSFER and SIEGE—by generating a lot more tests and achieving more exploits. ChatGPT worked better when prompts described more on the vulnerabilities, possible exploits, and code context. Our research will shed light on new research in security test generation. The generated tests will help developers create secure by design and secure by default software.

Current tools not secure

factor affecting attacks

Additional Key Words and Phrases: ChatGPT-4.0, supply chain attack, test generation, prompt design

1 INTRODUCTION

Chatgpt outperforms other tools, especially when prompted well

A supply chain attack, also called a value-chain or third-party attack, occurs when hackers infiltrate software systems through an outside partner or provider with access to those systems and data [7]. When such attacks target open-source software libraries, cybercriminals may compromise those libraries to distribute malicious code through the software supply chain, or leverage the known vulnerabilities in existing libraries to compromise any systems built on top of those libraries. In 2021, supply chain attacks grew 650%, and caused 12,000 incidents [6]. In 2022, there was a 742% year-over-year increase in open-source software supply chain attacks, targeting vulnerabilities in upstream ecosystems such as JavaScript, Java, .NET, and Python [49]. Open Web Application Security Project (OWASP) listed "vulnerable and outdated software components" as the sixth top vulnerability in 2022 [39].

Supply chain attacks are rapidly increasing

To mitigate supply chain attacks, researchers and engineers created a variety of tools to reveal vulnerable library dependencies in software applications [1, 2, 10, 34, 41, 42, 47, 77, 80], and even suggest fixes for those vulnerabilities [10, 17, 34, 73]. For instance, snyk-test [47] and npm-audit [34] 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),

---

Authors' addresses: Ying Zhang, yingzhang@vt.edu, Virginia Tech, Blacksburg, VA, USA, 24060; Wenjia Song, wenjia7@vt.edu, Virginia Tech, Blacksburg, VA, USA, 24060; Zhengjie Ji, zhengjie@vt.edu, Virginia Tech, Blacksburg, VA, USA, 24060; Danfeng (Daphne) Yao, danfeng@vt.edu, Virginia Tech, Blacksburg, VA, USA, 24060; Na Meng, nm8247@vt.edu, Virginia Tech, Blacksburg, VA, USA, 24060.

2023. XXXX-XXXX/2023/10-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Existing tools may scan libraries for dependencies that are in vulnerability databases

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

and report a vulnerability for each found match. Snyk-fix [17] is another CLI command, which eliminates vulnerabilities by automatically applying recommended updates to the vulnerable package dependencies revealed by snyk-test. or they update vulnerable dependencies

Tools do not show how they find vulnerabilities so they are rejected

Unfortunately, developers often do not trust the vulnerabilities reported by existing tools, because none of these tools demonstrate how the found vulnerabilities lead to serious consequences (e.g., denial of service) in their own projects [66]. As described in an article well received by developers *npm audit: Broken by Design* [4], “In many situations, (npm-audit) leads to a 99%+ false positive rate, creates an incredibly confusing first programming experience, ..., and at some point will lead to actually bad vulnerabilities slipping in unnoticed.” In recent studies [66, 88], **developers suggested that for reported vulnerable dependencies or vulnerable API usage, it is necessary to demonstrate how such vulnerabilities incur security attacks to their own systems**.

false results

goals

To (1) help developers mitigate the threats of supply chain attacks, and (2) persuade them into seriously considering the outputs of vulnerability detectors as well as fixers, we did a novel study to explore generating security tests for software applications (Apps) that depend on vulnerable libraries (Libs). These test cases demonstrate **proof-of-concept exploits**, which execute Apps in certain ways to (i) propagate vulnerabilities from Libs to Apps via calls of the APIs (i.e., functions) defined by Libs, and (ii) trigger abnormal behaviors of Apps such as throwing errors or becoming unresponsive to customers’ normal requests. When developers run these test cases, 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.

generating security tests is a better alternative that solves the issue

what we need

The biggest challenge in generating security tests is how to ensure that the generated tests (1) execute vulnerable API calls made by Apps, (2) trigger the problematic behaviors of Apps, and (3) fail when reported vulnerabilities are not fixed. Iannone et al. [64] recently created a tool SIEGE to generate security tests using EvoSuite [61]—the widely used test generation tool. Kang et al. [67] built TRANSFER, to generate tests by combining EvoSuite with program analysis techniques (e.g., call graph analysis and program instrumentation). Unfortunately, both tools fail to generate security tests most of the time. They often spend a lot of time producing irrelevant tests but cannot synthesize the specialized test inputs, code, or oracle to demonstrate proof-of-concept exploits. Based on our initial experience with ChatGPT [29]—an artificial intelligence chatbot developed by OpenAI, we found that ChatGPT is able to create programs to satisfy our software requirements described in English. Thus, in this project, we explored ChatGPT’s capability in generating security tests for Apps with vulnerable dependencies.

current tools fail

This paper explores chatgpt generating tests

We explored the following research questions (RQs) and observed interesting phenomena:

RQ1: *How effectively does ChatGPT generate security tests?* We created a dataset to include (1) 26 Libs and (2) 55 Apps, with each App depending on a vulnerable library version. For each App, we offered ChatGPT a prompt to describe the vulnerability, program context in App, and a security test from Lib showing a proof-of-concept exploit. 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 55 Apps, 24 of which effectively demonstrated the proof-of-concept exploits. mimicks really well

RQ2: *How does ChatGPT’s security 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 more important than the others. Without security test from Lib provided, none of the generated tests by ChatGPT could successfully exploit any vulnerability. but needs guidance

RQ3: *How does ChatGPT compare with existing tools of security test generation?* We also applied two state-of-the-art tools—SIEGE and TRANSFER—to our dataset. Surprisingly, we observed ChatGPT

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

much better than Seige and transfer

to outperform both tools. Given the 55 apps, ChatGPT exploited the vulnerabilities in 24 apps; TRANSFER exploited 4 vulnerabilities; SIEGE exploited none.

2 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 [48]. According to CVE-2020-28052 [20], its releases 1.65 and 1.66 have a vulnerability: `OpenBSDBCrypt.checkPassword(...)` compares incorrect data when checking the password, allowing wrong passwords to be accepted as valid ones.

BC are Java APIs used here as an example

Listing 1 shows the security test defined by a BC version later than 1.66, which demonstrates the vulnerability and the proof-of-concept exploit. As shown in Listing 1, ideally, if a BC version has no vulnerability, the first assertion (line 10) should succeed as the first parameter `tokenString` was derived from the password `test-token`; the second assertion (line 11) should succeed 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).

first listing should succeed (TP) second should fail (TN)

Listing 1. A security test to demonstrate the proof-of-concept exploit of the vulnerability in BC 1.65 & 1.66

```
1  public 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.
9          generate("test-token".toCharArray(), salt, costFactor);
10     isTrue(OpenBSDBCrypt.checkPassword(tokenString, "test-token".toCharArray()));
11 /* A safe BC version should pass the following assertion. A vulnerable BC version should fail
        the assertion, as it treats unmatched passwords as matching ones. */
12     isTrue(!OpenBSDBCrypt.checkPassword(tokenString, "wrong-token".toCharArray()));
13   } }
```

Not very clear for me

yet not very useful

Although Listing 1 shows a proof-of-concept exploit of the library vulnerability, it does not show how Apps built on top of vulnerable BC versions can behave abnormally or get attacked due to that vulnerability. In the scenarios when an App does not call `OpenBSDBCrypt.checkPassword(...)`, the App is not influenced by the vulnerability at all; consequently, even if existing dependency checkers (e.g., OWASP Dependency-Check [2]) report a vulnerable dependency for App, the report is not useful but can mislead or irritate developers.

Our research intends to generate security test cases for Apps, in order to demonstrate (1) how vulnerabilities get propagated from Libs to Apps, and (2) how the resulting vulnerabilities in Apps can be leveraged by hackers.

missing

3 METHODOLOGY

We did an empirical study on ChatGPT's capability of generating security test cases, for Apps which have dependencies on vulnerable Libs and call vulnerable APIs. Our study has three phases: dataset construction (Section 3.1), prompt design (Section 3.2), and result validation (Section 3.3). Phase I collects known vulnerabilities in Libs, exemplar tests that exploit vulnerabilities, and Apps that can be affected by their usage of vulnerable APIs. Phase II 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, and to generate security tests

gather vuls and info about them

prompt chatgpt for vuls

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

that demonstrate proof-of-concept exploits. Phase III gathers all outputs by ChatGPT if there is any, assesses the quality of generated tests, and evaluates ChatGPT's capability accordingly.

assess results

3.1 Phase I: Dataset Construction

We took two steps to create a dataset: (1) finding exploitable vulnerabilities, (2) getting vulnerable Libs and dependent client Apps.

use datasets with known vuls

3.1.1 Locating Exploitable Vulnerabilities. As vulnerabilities sparsely exist in software libraries, it can be very inefficient to blindly mine GitHub repositories for vulnerabilities. Thus, we started with the datasets mentioned by prior work [67, 78] and initiated our exploration with 628 entries. Each entry is linked to a CVE entry or JIRA issue to describe a vulnerability in a Java library *Lib*, and a GitHub repository that shows both the vulnerable and patched versions of *Lib*.

relied on patches to understand vuls

We manually inspected all available data for each entry, to decide what security test was added in the patched version, what APIs were revised by the patched version, and how the security test(s) as well as revised APIs are relevant to the described vulnerability. Because vulnerability description is not always precise or comprehensive in pinpointing the vulnerable APIs in Libs, we spent lots of time understanding the program context to locate vulnerable APIs. We consider an API in *Lib* to be vulnerable if it (1) is mentioned or implied by the vulnerability description of CVE or JIRA entry and is 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). Afterwards, we chose vulnerability entries based on the following criteria: Did some filtering for Libs based on:

Had to do manual work due to unclarity

Vul criteria

(a) The Java library *Lib* has at least one JUnit test from the patched version, to demonstrate behavioral differences between the vulnerable and patched versions. Can be easily exploited
(b) *Lib* should compile successfully. Can't be tested to show diff behavior
(c) The test execution does not require for complex setups of client/server machines.

Criterion (a) ensures the exploitability of confirmed vulnerabilities. Namely, if no security test is included for a given vulnerability, it is hard for us to manually craft or justify the ground truth of proof-of-concept exploit. Criteria (b) and (c) ensure that we can run the security test defined for *Lib*, to observe the behavioral differences between vulnerable and patched versions. In our study, we treated the three criteria mentioned above as filters, and applied them one-by-one to refine initial vulnerability datasets. In particular, Criterion (a) removed most entries (i.e., 427) from the original datasets: 304 entries were removed because Java libraries have no test, and 123 entries were removed because the tests are irrelevant to vulnerabilities. Criterion (b) eliminated 49 entries, and (c) removed additional 107 entries. At the end of this step, we found 45 unique entries to match all criteria mentioned above.

started with 628

finding apps built on libs

3.1.2 Collecting Libs and Apps for Vulnerabilities. For each *Lib* mentioned in the 45 refined vulnerability entries (see Section 3.1.1), we searched on GitHub for client Apps depending on *Lib*, by using the library or package name as keyword(s). As GitHub typically retrieved lots of projects for each of our keyword-based search requests, we could not afford the time or effort of examining all projects. Thus, for each request, we limited ourselves to inspect the first 10 pages of results returned by GitHub, aiming at finding up to 4 client applications to satisfy the criteria below:

(d) At least one non-private Java method (not test) in *App* (in)directly calls vulnerable API(s) in *Lib*, with non-constant parameters.
(e) *App* compiles and runs successfully. App uses the vul api and runs

Criterion (d) ensures the feasibility of security exploits. Basically, if users of *App* craft malicious input values to feed certain public or protected method(s) in *App* (i.e., the callers of vulnerable

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

APIs), they can run vulnerable APIs in malicious ways and thus realize attacks. Criterion (e) ensures that we can check the correctness of tool-generated tests via compilation and program execution.

Table 1. 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	XStream [52]	[, 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	Jackson-Modules-Java8 [25]	[, 2.9.8)	`ObjectMapper.readValue(...)` triggers DoS when it deserializes a very large decimal value to time.	4
	CVE-2018-11761	Apache Tika [16]	[0.1, 1.18]	`SAXParser.parse(...)` was not configured to limit entity expansion, and thus could lead to DoS.	1
	CVE-2018-12418	Junrar [30]	[,1.0.1)	The `Archive` constructor gets into an infinite loop when handling corrupt RAR files.	1
	CVE-2018-1274	Spring Data Commons [43]	[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	Apache Tika	[1.19, 1.21]	`Parser.parse(...)` enables a carefully crafted 2003ml or 2006ml file to consume all available SAXParsers in the pool.	3 (2*)
	CVE-2019-12402	Apache Commons Compress [19]	[1.15, 1.18]	Malicious inputs to `ZipArchiveOutputStream.putArchiveEntry(...)` or `ZipEncoding.encode(...)` can cause infinite loops.	1
	CVE-2020-28491	Jackson Dataformat: CBOR [24]	[, 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	Json-smart [32, 33]	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	Apache CXF [14]	[, 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	JSON-java(i.e., hutool-json) [46]	[, 20230227)	Malicious inputs to `XML.toJSONObject(...)` or `JSONML.toJSONObject(...)` can trigger DoS.	3
	TwelveMonkeys-595	TwelveMonkeys [26]	[0, 3.6.4)	A corrupt JPEG file to `ImageReader.read(...)` can cause DoS.	2 (1*)
	Zip4j-263	Zip4j [45]	[0, 2.7.0)	The `ZipFile(...)` constructor can take in a null File reference, which later produces a null pointer exception.	2
Directory Traversal (6)	CVE-2018-1002200	Plexus Archiver [40]	[,3.6.0)	`UnArchiver.extract(...)`, `ZipUnArchiver.extract(...)`, and `TarGZipUnArchiver.extract(...)` allow attackers to write to arbitrary files via "../" in an archive entry (**Zip Slip**).	3
	CVE-2018-1002201	ZT Zip [53]	[, 1.13)	`ZipUtil.unpack(...)` allows attackers to write to arbitrary files via archive extraction (Zip Slip).	1
	CVE-2018-19859	OpenRefine [36]	[, 3.2-beta)	`ImportingUtilities.allocateFile(...)` allows arbitrary file write via archive extraction (Zip Slip).	1(1*)
	CVE-2018-20227	RDF4J [22]	2.4.2	`ZipUtil.extract(...)` enables arbitrary file write via archive extraction (Zip Slip).	1
	CVE-2021-29425	Apache Commons IO [13]	[, 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	Jackson Databind [23]	[, 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	XStream	[, 1.4.14)	Malicious inputs to `XStream.fromXML(...)` allow attackers to run arbitrary shell commands.	3

Many of them result in DoS

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

	CVE-2021-23899	OWASP JSON Sanitizer	[,1.2.2)	`JsonSanitizer.sanitize(...)` may allow hackers to inject arbitrary code into embedding documents.	1
	CVE-2022-25845	Fastjson [11]	[, 1.2.83)	`JSON.parseObject(...)` may deserialize untrusted data, allowing hackers to attack remote servers.	2
Others (7)	CODEC-134	Apache Commons Codec [18]	[, 1.13)	Malicious inputs to `Base64.decodeBase64(...)` or `Base64.decode(...)` can realize **covert channel** [87], which creates a capability of transferring data between processes that should not communicate	3
	CVE-2018-1000632	Dom4j [21]	[, 2.1.1)	Malicious inputs to `DocumentHelper.createElement(...)` or `Branch.addElement(...)` can result in **XML injection**, which tampers with XML documents.	3
	CVE-2020-13956	Apache Http-Client [15]	[, 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	OWASP JSON Sanitizer [38]	[,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	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	Spring Security [44]	[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	snappy-java [51]	[, 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.

Some of them are more malicious

more filtration

Such a manual crawling process removed 15 from the 45 entries mentioned above, because we found no client project to satisfy both criteria for those entries. As shown in Table 1, our dataset includes 30 vulnerability entries, corresponding to 26 unique libraries. These libraries cover various domains, such as data processing (e.g., Apache Commons Codec [18]), web development (e.g., Apache CXF [14]), and security (e.g., Spring Security [44]). Most libraries have single vulnerabilities (e.g., Dom4j [21]), while a few libraries have multiple (e.g., XStream [52]). In Table 1, We identified 4 major categories among the 30 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. dataset construction

domains

Vuls

According to our experience, it is very challenging to crawl for sufficient Apps satisfying (e)–(f) for any vulnerable library. To conduct a representative empirical study with sufficient data points, we had to fully leverage the retrieved Apps even though they do not depend on vulnerable library versions. Specifically for each found project *App* satisfying criteria (e)–(f), we examined the version history to reveal any version of *App*—e.g., App_i—that depends on a vulnerable library version. If App_i exists, we checked out App_i to prepare for ChatGPT usage. Otherwise, we manually revised dependency to inject the vulnerability. For instance, OpenRefine [36] 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 to 3.1 and tried to compile the revised project.

check old versions OR inject vul

example

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 1 shows

Injecting does not affect results

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

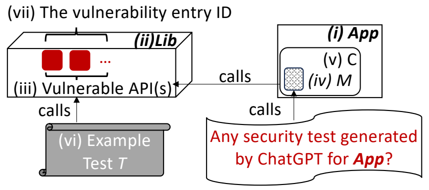


Fig. 1. Our default usage of ChatGPT to generate security tests

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 prompt template

the number of clients we included for each Lib. Our dataset includes in total seven projects with injected vulnerable dependencies, all of which are marked with asterisks (*) in the table.

Notice that we do not expect ChatGPT to automatically detect vulnerable dependency or the usage of vulnerable APIs. Instead, we assume some tools or domain experts to detect vulnerabilities, and provide all relevant data to ChatGPT for exploit generation. Our experiment simulates such a usage scenario of ChatGPT, and our dataset has the data below for each exploit generation task:

- **(i)** A GitHub project or software application *App*.
- **(ii)** A vulnerable version of library *Lib* on which *App* depends.
- **(iii)** The vulnerable library API(s) called by *App*.
- **(iv)** The non-private method *M* inside *App* that (in)directly calls vulnerable APIs.
- **(v)** The Java class *C* defining method *M*.
- **(vi)** The exemplar security test case *T* (i.e., a Java method) from *Lib*. 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 ID or bug issue ID). Description of the process (imp)

Fig. 1 illustrates the relations between information items (i)–(vii) mentioned above. Basically, a vulnerable version of the library *Lib* can include one or more vulnerable APIs; such a vulnerability is explained in detail by a CVE report or bug issue, which can be uniquely referenced or identified via a vulnerability entry ID. A later version of *Lib* may fix the vulnerability, and define an exemplar security test *T* to provide maliciously crafted inputs as it calls vulnerable API(s). In this way, *T* can demonstrate the behavioral differences between the vulnerable and fixed versions, or show a proof-of-concept exploit of the known vulnerability. Meanwhile, the application *App* depends the vulnerable library version and defines a method $C.M(\dots)$ to call the vulnerable API(s).

At the end of Phase I, our dataset consists of all relevant information for 55 $\langle App, Lib \rangle$ pairs. We leveraged that information to define 55 exploit generation tasks, which were used in later phases.

3.2 Phase II: Prompt Design

Based on the information collected in Phase I, we formulated prompts, and sent those prompts to ChatGPT to generate exploits for Apps with vulnerable dependencies. This section introduces both our default template design for prompts (Section 3.2.1) and alternative designs (Section 3.2.2).

3.2.1 The Design of Our Default Prompt Template. As shown in Fig. 1, we typically used ChatGPT to generate a security test for any given $\langle App, Lib \rangle$ pair, by providing as much relevant information as possible in the prompts. We intended to check whether ChatGPT can (1) mimic the exemplar test *T*, and (2) similarly craft malicious inputs to *M* so that *M* also calls library API(s) with malicious

Can chatgpt: ① Mimic given T
② Generate malicious inputs

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

used gathered data to generate the prompts

inputs. To do that, we defined a prompt template that can be concretized with the information (iii)–(vii) mentioned in Section 3.1.2. As shown in Fig. 2, the template 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)**. Using this template, we generated 55 prompts for our dataset, and sent all prompts to ChatGPT.

Listing 2. A GitHub project defines a Java file to call `OpenBSDBCrypt.checkPassword()`

```java
public class BcryptPasswordHashFunction implements PasswordHashFunction {
  ...
// The method below calls the vulnerable API, so it can be affected by CVE-2020-28052.
  @Override
  public boolean check(String passwordHash, String password) {
    // the vulnerable API call
    return OpenBSDBCrypt.checkPassword(passwordHash, password.toCharArray());
  }
  ...
}
```

For the motivating example described in Section 2, 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 proof-of-concept exploit for the client method `BcryptPasswordHashFunction.check(...)`, we formulated the prompt shown in Fig. 3 by customizing our template. This figure only shows partial code of example test and client class to simplify our presentation, while the actual prompt we sent to ChatGPT includes the complete code.

Sample prompt

~~~~
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: [OpenBSDBZCrypt.checkPassword]. Mimic the
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
@Test
public void performTest() throws Exception {
  ... // Here we omit details of BC security test (Listing 1) for brevity.
}
```
CLIENT CODE:
``` java
public class BcryptPasswordHashFunction implements PasswordHashFunction {
  // Here we omit details of BcryptPasswordHashFunction.java, which defines the method check(...) in Listing 2
  ...
}
```
~~~~

Fig. 3. A prompt derived from our template

task 1 done

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 the vulnerability exploit for client code.

Could be a weakness but achieves goal none the less

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

```java
... // We omit less important details for brevity
public class BcryptPasswordHashFunctionTest {
```

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

```
  ...
  @Test
  public void testCheckFunction() {
    int costFactor = 4;

    for (int i = 0; i < 1000; i++) {
      random.nextBytes(salt);
      final String tokenString = OpenBSDBCrypt.generate("test-token".toCharArray(), salt,
           costFactor);
      assertTrue(bcryptPasswordHashFunction.check(tokenString, "test-token"));
/* The App should fail the following assertion, when it depends on a vulnerable BC version
   that messes up correct with incorrect passwords.*/
      assertFalse(bcryptPasswordHashFunction.check(tokenString, "wrong-token"));
} } }
```

Table 2. The six prompt templates we explored

omitted parts of the prompt for every test

Id	Prompt Template	Id	Prompt Template
P	Default (including all elements 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)

3.2.2 *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 elements contribute to ChatGPT's effectiveness. As shown in Table 2, to facilitate presentation, we use P to simply refer to our default template design, which leverages the information elements (iii)–(vii). We use P_1–P_5 to separately refer to the five template variants, each of which contains one fewer item than P. For instance, P_1 leaves out (iii), but defines four variables to take in (iv)–(viii). P_5 leaves out (vii), but takes in (iii)–(vi). We then generated 55 prompts using each variant template, and sent all prompts to ChatGPT.

3.3 Phase III: Result Validation

After sending all prompts to ChatGPT, we gathered and recorded ChatGPT's outputs if there is any. For each test class T' generated by ChatGPT, we copied the code to its corresponding software project App. If T' has a unique name and defines one or more test functions for method-to-test M, we put the class into the test folder of App. Otherwise, if T' has the same name as an existing test class, we copied the relevant non-conflicting content and appended it to that class; if T' defines test functions for a non-public method M, we copied the code and pasted it to any existing (test) class where M is accessible. To sum up, we integrated the generated tests into client projects, ensuring that the tests were put at the right places.

fixed simple errors manually

For each generated test, we compiled the whole project App to examine 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 was able to synthesize the most important logic of security tests successfully. After one or multiple iterations of the compilation-and-fixing procedure, if a test compiled successfully, we further executed the whole project 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 vulnerability exploitation among authors until reaching a consensus.

4 EXPERIMENTS AND RESULTS

We did experiments for the following research questions (RQs):

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

Table 3. Security test generation by ChatGPT (Total: 55 A, 40 C, 24 V)

Idx	Vulnerability Entry ID	# of Clients	A	C	V	Idx	Vulnerability Entry ID	# of Clients	A	C	V
1	CODEC-134	3	3	2	2	16	CVE-2020-13973	1	1	0	0
2	CVE-2017-7525	2	2	2	2	17	CVE-2020-26217	3	3	3	2
3	CVE-2017-7957	2	2	1	1	18	CVE-2020-28052	2	2	2	1
4	CVE-2018-1000632	3	3	1	1	19	CVE-2020-28491	1	1	1	0
5	CVE-2018-1000873	4	4	2	1	20	CVE-2020-5408	2	2	2	2
6	CVE-2018-1002200	3	3	1	0	21	CVE-2021-23899	1	1	1	0
7	CVE-2018-1002201	1	1	1	1	22	CVE-2021-27568	2	2	2	2
8	CVE-2018-11761	1	1	0	0	23	CVE-2021-29425	3	3	2	2
9	CVE-2018-12418	1	1	0	0	24	CVE-2021-30468	1	1	1	0
10	CVE-2018-1274	1	1	0	0	25	CVE-2022-25845	2	2	2	2
11	CVE-2018-19859	1	1	1	0	26	CVE-2022-45688	3	3	3	1
12	CVE-2018-20227	1	1	1	0	27	CVE-2023-34454	1	1	1	1
13	CVE-2019-10093	3	3	2	0	28	HTTPCLIENT-1803	1	1	1	0
14	CVE-2019-12402	1	1	0	0	29	TwelveMonkeys-595	2	2	2	0
15	CVE-2020-13956	1	1	1	1	30	Zip4j-263	2	2	2	2

RQ1: *How effectively does ChatGPT generate security tests?* We investigated the strengths and weaknesses of ChatGPT when it generates proof-of-concept exploits for known vulnerabilities.

RQ2: *How does ChatGPT's security performance differ given various types of prompts?* Among the various information we provided to ChatGPT (see (iii)–(vii) mentioned in Section 3.1.2), we wanted to learn which type of information is more crucial, and how each information type contributes to ChatGPT's capability of security test generation.

RQ3: *How does ChatGPT compare with existing tools of security test generation?* This RQ explores whether it is worth the effort of trying to generate security tests using large language models (LLMs), instead of using program analysis techniques.

This section first introduces the metrics we defined to assess tools of security test generation (Section 4.1). It then explains our experiments and results for the RQs (Sections 4.2–4.4).

4.1 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.

Vulnerability Exploitation (V) counts the number of compilable tests that exploit the known vulnerabilities as expected.

4.2 ChatGPT's Effectiveness in Security Test Generation

We created 55 prompts using the default prompt template *P* (see Section 3.2), and sent them to ChatGPT for results. This experiment examines "*When we provide as much relevant information as possible in the prompts sent to ChatGPT, how effectively can it generate security tests?*" As shown in Table 3, ChatGPT generated tests for all ? prompts: 26 of the tests are compilable and runnable; 14 tests are compilable and runnable after we manually applied minor fixes (e.g., customizing hardcoded file paths); 24 of these 40 tests (26 + 14) effectively mimic the behaviors of given library tests, and successfully exploit relevant errors or runtime exceptions.

Mimics existing tests almost perfectly

4.2.1 Uncompilable Tests. Among the 55 generated tests, 15 tests do not compile and cannot get easily fixed via minor changes. Six tests do not compile as they violate Java access rules. For instance, four of the tests access private members from outside of the classes (i.e., fields and methods); one test directly references a method defined inside an enum from outside; one test has an ambiguous method reference, which can be interpreted as a call to either of two same-named methods. Another 5 of the 15 tests use undefined program entities such as methods and classes. Finally, 4 of the 15

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

Most tests generated compiled but not all of them

tests call methods with inappropriate parameter lists, by missing some parameters or using wrongly typed parameters. Our observations imply that ChatGPT does not guarantee code compilation, even though the majority of tests it generated (40/55) are easy to compile.

compiled but ineffective

4.2.2 Ineffective or Less Effective Tests. Sixteen 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. Four reasons can explain such ineffectiveness.

Some issues with the generated compiled tests

(1) Mockito [31]—a mocking framework—was frequently used by ChatGPT to mock unknown variables, methods, or classes when generating tests. Unfortunately, the framework could not mock everything (e.g., final classes). As a result, it led to MockitoExceptions thrown by some of the tests.
(2) In generated tests, the parameters passed to the method-to-test 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.
(3) Some client projects already applied patches in response to the known library vulnerabilities. Thus, even though the generated tests try to trigger those vulnerabilities, the client projects screen out those trials and prevent library vulnerabilities from being exploited.
(4) An unexpected bug in a project was revealed by a generated test. This bug causes a runtime exception and prevents the test from further execution to trigger the known vulnerability CVE-2021-23899. We actually filed an issue for the newly revealed bug, and developers of that project confirmed our bug report.

4.2.3 Effective Tests. Twenty-four generated tests can trigger vulnerabilities as expected. For all of these tests, ChatGPT successfully extracted vulnerability-triggering inputs from the exemplar tests, reused those inputs in test generation, prepared meaningful values for parameters to call method M, and threw exactly the expected exceptions/errors or presented relevant abnormal program behaviors (e.g., infinite loop and timeout). Based on our experience, when M calls vulnerable API(s) directly, requires for very few and simple parameters, and has simple program logic without many conditional or loop statements, ChatGPT was more likely to generate security tests successfully. Meanwhile, when M calls vulnerable API(s) indirectly, requires for many or complex parameters, or has complex program logic with a lot of code, ChatGPT was unlikely to generate good tests.

important inter-pretation

Finding 1: *ChatGPT is promising in generating security tests for known library vulnerabilities. Given 55 test generation tasks, it produced 55 tests, 40 of which are easy to compile and 24 of the tests successfully exploited vulnerabilities.* First result

4.3 Information Elements That May Impact ChatGPT's Effectiveness

We used the 5 template variants P_1–P_5 mentioned in Section 3.2, generated prompts for the 55 $\langle App, Lib \rangle$ pairs in our dataset, and sent them to ChatGPT for results.

Table 4. The comparison of applicability and compilability between tests generated in different ways

Id	Prompt Template	Tool Applicability (A)	Test Compilable?			
			Yes (C)	No		
				Access Rule Violation	Incorrect Method Calls	Unknown Entity Usage
P	Default (all elements)	55	40	6	4	5
P_1	Without vulnerable APIs (iii)	55	39	3	2	11
P_2	Without M (iv)	55	42	5	1	7
P_3	Without C (v)	55	16	3	1	35
P_4	Without the exemplar test (vi)	55	32	7	2	14
P_5	Without the vulnerability ID (vii)	55	40	8	0	7

Java class defining M

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

4.3.1 ChatGPT's Applicability Given Divergent Types of Prompts. As shown in Table 4, 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.

4.3.2 ChatGPT's Test Compilability Given Divergent Types of Prompts. The number of compilable tests for each prompt template varies a lot. As shown in Table 4, when P_2 was used and M was not specified, ChatGPT generated more compilable tests than what it did for the default template P (42 vs. 40). When P_5 was used and no vulnerability ID was mentioned, ChatGPT generated the same number of compilable tests as what it did for P—40. When P_1 was used and the vulnerable APIs were not specified, ChatGPT generated slightly fewer compilable tests—39. However, when P_3 and P_4 were used, a lot fewer generated tests compile, i.e., 16 and 32. One possible reason is that both P_3 and P_4 significantly removed the code context relevant to test generation, 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 [28]. It tended to generate more compilable tests when more relevant program context was provided. needs context

Among the six prompt templates, P_3 caused ChatGPT to create the most uncompilabe tests—39; 35 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—13, only 7 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. } Conclusion

One reason for failure

No matter what template we tried, ChatGPT always produced uncompilable tests for some prompts. This indicates that ChatGPT does not strictly follow the rules of Java program syntax and semantics. As a general AI model, it is trained on a massive dataset of text from the Internet; it leverages its training to predict what words and phrases are likely to come next in a given context. 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 2:** *Among the five template variants we explored, P_3 and P_4 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.*

Results for exploiting vul

Table 5. The comparison of vulnerability exploitation between tests generated in different ways

Idx	Prompt Template	Vulnerability Exploited?			
		Yes (V)	No		
			No error/exception	Mockito exception	Other exceptions/errors
P	Default (all elements)	24	4	2	10
P_1	Without vulnerable APIs (iii)	15	6	8	10
P_2	Without M (iv)	14	4	8	16
P_3	Without C (v)	1	7	5	3
P_4	Without the exemplar test (vi)	0	11	11	10
P_5	Without the vulnerability ID (vii)	14	5	5	16

4.3.3 ChatGPT's Vulnerability Exploitation Given Divergent Types of Prompts. As shown in Table 5, removing any element from P made ChatGPT exploit vulnerabilities less effectively. Specifically, among the five variants, P_4 caused ChatGPT to work worst, producing zero successful vulnerability

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

exploit. This implies that exemplar tests offer (1) crucially important demonstration for potential exploit methods, and (2) essential hints for ChatGPT to formulate vulnerability-triggering input values. Without such information, ChatGPT generated 11 tests throwing no error/exception, 11 tests wrongly mocking program entities, and 10 tests triggering irrelevant errors or exceptions. Although slightly better than P_4, P_3 also worsened ChatGPT significantly and led it to generate only one exploit successfully. This may be because the removed Java class C holds lots of program context information, whose absence caused ChatGPT to produce tests in a context-insensitive way, making the produced 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 successful exploits 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.

Finding 3: *Among the five information elements covered by the default prompt template P, all elements played an important role to help ChatGPT effectively generate vulnerability exploits. In particular, (v) and (vi) were more important than (iii), (iv), and (vii).*

4.4 Tool Comparison

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, it requires users to describe the search target (i.e., the coverage goal for tests-to-generate), including (1) fully qualified name of the target vulnerable class in *Lib*, (2) vulnerable API method name, and (3) vulnerable line number. SIEGE reuses EvoSuite [61]—the popularly used test generation tool—to randomly generate tests, select tests based on their closeness to the specified target, and evolve those tests with some randomness to generate better tests that are closer to the target. SIEGE stops when the time budget is used up or some tests perfectly match the target. Smart random algorithms

Similar to SIEGE, TRANSFER also tries to generate security tests using GA, adopts the program analysis feature of EvoSuite to create both call graphs and control-flow graphs for *App*, and leverages the dynamic instrumentation feature of EvoSuite to assess test coverage. However, for any $\langle App, Lib\rangle$ pair, TRANFER requires users to specify (1) the vulnerable API and (2) a library test to show the vulnerability exploit. TRANSFER then dynamically instruments *Lib*, to identify the program states relevant to the vulnerability, and to extract the conditions that must be satisfied by any security test to show the proof-of-concept exploit. Finally, TRANSFER adopts the extracted information to guide EvoSuite and to generate security tests. Due to the adoption of exemplar security test from *Lib* and more advanced program analysis techniques, TRANSFER manifested better effectiveness than SIEGE in prior work [67]. Not very clear

4.4.1 Experiment. We prepared the inputs required by SIEGE and TRANSFER for 55 $\langle App, Lib\rangle$ pairs, and executed both tools accordingly. For each test output by either tool, we copied the code and pasted it to appropriate places. We leveraged the build process to reveal compilation errors and applied minor fixes to obvious and simple compilation errors, ensuring to fairly compare the outputs by different tools. If compilation succeeded, we further executed the project with the generated 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 among authors until reaching a consensus.

Did the same process as before

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

Table 6. The comparison between ChatGPT and state-of-the-art security test generators

much better performance

	Tool Applicability (A)	**Test Compilability (C)**	**Vulnerability Exploitation (V)**
ChatGPT	55	40	24
TRANSFER	16	13	4
SIEGE	1	1	0

4.4.2 Results. As shown in Table 6, surprisingly, ChatGPT outperformed both state-of-the-art tools considerably by having much better tool applicability, test compilability, and vulnerability exploitation. Specifically, ChatGPT generated tests for all 55 ⟨*App*, *Lib*⟩ pairs, while TRANSFER only generated test functions or code snippets for 16 pairs. SIEGE worked much worse, generating tests for only one pair.

In terms of compilability, 13 out of the 16 tests generated by TRANSFER are compilable, while 3 tests fail to compile due to their usage of unknown entities. The only test output by SIEGE compiles successfully. Based on the small number of tests generated by TRANSFER and SIEGE, it is hard to conclude whether they were more likely to generate compilable tests than ChatGPT.

In terms of vulnerability exploitation, only four of the tests output by TRANSFER successfully trigger vulnerabilities; the remaining nine compilable tests fail to do so. Among these ineffective or less effective ones, six tests execute smoothly, without triggering 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. For the four tasks handled well by TRANSFER, ChatGPT also successfully triggered vulnerabilities. Our observations imply that ChatGPT worked overwhelmingly better than TRANSFER and SIEGE.

Several reasons can help explain why TRANSFER and SIEGE worked worse than ChatGPT in security test generation. First, both tools adopt EvoSuite, to generate tests and execute the method-to-test *M*. However, EvoSuite is not always effective; some or even most of the tests generated by EvoSuite do not execute *M* at all. Second, both tools have difficulty synthesizing or mocking complex input parameters for *M*. For instance, neither tool synthesized a parameter of type `net.sourceforge.pmd.RuleSet`, but ChatGPT mocked such an object via the Mockito framework.

Third, SIEGE cannot take in any exemplar test to infer the exploit method or vulnerability-triggering inputs, and it has no domain knowledge about security exploits. Meanwhile, although TRANSFER can take in exemplar tests to infer the exploit knowledge, it has difficulty incorporating inferred information into test generation. For instance, CODEC-134 is related to vulnerable APIs `Base64.decode(...)` and `Base64.decodeBase64(...)`. As shown in Fig. 4, we provided both TRANSFER and ChatGPT inputs relevant to that vulnerability, including the exemplar library test to demonstrate the security exploit of `Base64.decode(...)` (see Fig. 4 (a)), and a client Java class that calls the vulnerable API `Base64.decodeBase64(...)` (see Fig. 4 (b)). We tried both tools to generate a security test for the client code, to trigger the vulnerability similarly as the exemplar test. In the figure, (c) and (d) separately show the tools' outputs. Unfortunately, TRANSFER could not reuse any domain knowledge demonstrated by the exemplar test. However, ChatGPT successfully transferred the domain knowledge and effectively produced a security test.

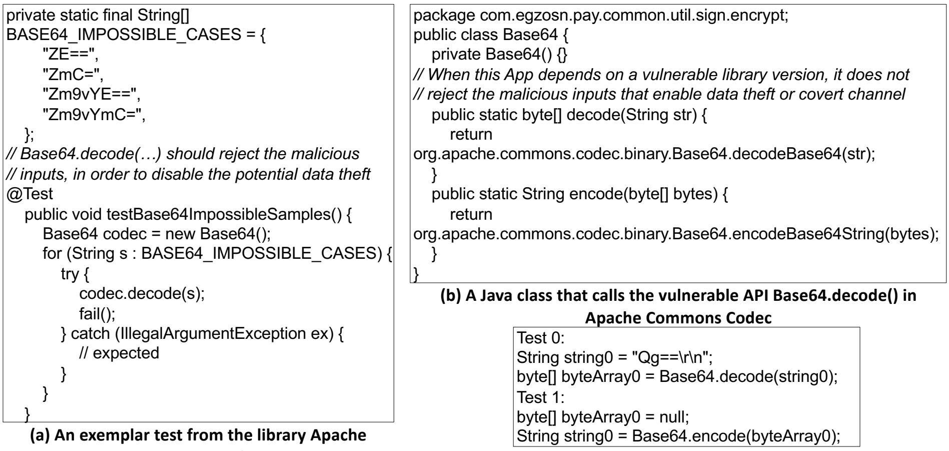
Finding 4: *ChatGPT outperformed both TRANSFER and SIEGE in terms of all metrics we evaluated: tool applicability, test compilability, and vulnerability exploitation. This observation implies the great potential of ChatGPT in security test generation.*

5 THREATS TO VALIDITY

Could be limited generalizability

Threats to External Validity: All our observations are limited to the vulnerabilities, libraries, and client projects included into our dataset. Our study focuses on Java programs, as we have more experience and domain knowledge relevant to the programming language and software projects;

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com



```
private static final String[]
BASE64_IMPOSSIBLE_CASES = {
        "ZE==",
        "ZmC=",
        "Zm9vYE==",
        "Zm9vYmC=",
    };
// Base64.decode(…) should reject the malicious
// inputs, in order to disable the potential data theft
@Test
    public void testBase64ImpossibleSamples() {
        Base64 codec = new Base64();
        for (String s : BASE64_IMPOSSIBLE_CASES) {
            try {
                codec.decode(s);
                fail();
            } catch (IllegalArgumentException ex) {
                // expected
            }
        }
    }
```

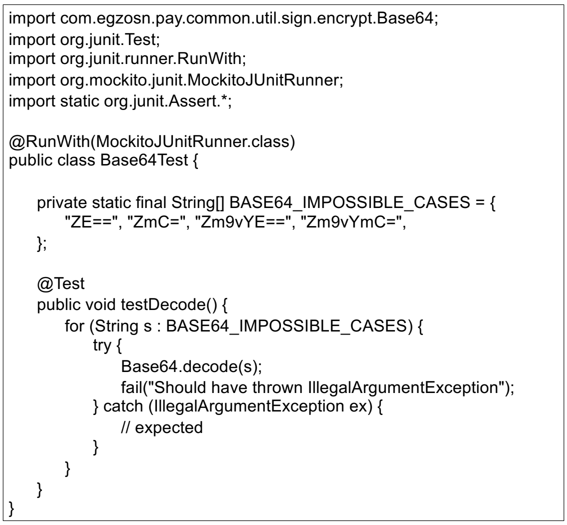
(a) An exemplar test from the library Apache Commons Codec

```
package com.egzosn.pay.common.util.sign.encrypt;
public class Base64 {
    private Base64() {}
// When this App depends on a vulnerable library version, it does not
// reject the malicious inputs that enable data theft or covert channel
    public static byte[] decode(String str) {
        return
org.apache.commons.codec.binary.Base64.decodeBase64(str);
    }
    public static String encode(byte[] bytes) {
        return
org.apache.commons.codec.binary.Base64.encodeBase64String(bytes);
    }
}
```

(b) A Java class that calls the vulnerable API Base64.decode() in Apache Commons Codec

```
Test 0:
String string0 = "Qg==\r\n";
byte[] byteArray0 = Base64.decode(string0);
Test 1:
byte[] byteArray0 = null;
String string0 = Base64.encode(byteArray0);
```

(c)Two tests generated by TRANSFER that fail to exploit the Apache Commons Codec vulnerability CODEC-134



```
import com.egzosn.pay.common.util.sign.encrypt.Base64;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.junit.MockitoJUnitRunner;
import static org.junit.Assert.*;

@RunWith(MockitoJUnitRunner.class)
public class Base64Test {

    private static final String[] BASE64_IMPOSSIBLE_CASES = {
        "ZE==", "ZmC=", "Zm9vYE==", "Zm9vYmC=",
    };

    @Test
    public void testDecode() {
        for (String s : BASE64_IMPOSSIBLE_CASES) {
            try {
                Base64.decode(s);
                fail("Should have thrown IllegalArgumentException");
            } catch (IllegalArgumentException ex) {
                // expected
            }
        }
    }
}
```

(d) The test generated by ChatGPT that successfully exploits the Commons Codec vulnerability CODEC-134

Fig. 4. An example to show the output difference between TRANSFER and ChatGPT

we did not apply ChatGPT to programs written in other languages. So far, we have only queried ChatGPT to generate JUnit tests to demonstrate security exploits. We did not query ChatGPT to generate more diverse test cases for integration testing or system testing. Our dataset does not include more complex vulnerable scenarios, which require for network configurations or client-server application setups. Therefore, our observations may not generalize well to other vulnerabilities, other software, other programming languages, or other testing methods. In the future, to make our research findings more representative, we will expand our dataset to cover more vulnerabilities, more programs written in different languages, and more applications involving complex configurations. Study is insightful yet limited

Threats to Internal Validity: We experimented with the default setting of ChatGPT, without controlling or tuning any parameter it defines. By default, when ChatGPT is queried with the same prompt multiple times, it generates results with randomness, i.e., it can produce different results given the same prompt. Such randomness can impact the validity or certainty of our observations.

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

Room for other studies? <– Python
Tuning Chatgpt params

However, based on our experience so far, ChatGPT often produces very similar results given multiple trials of the same prompt. We believe that the internal randomness of ChatGPT does not significantly impact our experiment results.

Threats to Construct Validity: ChatGPT was trained on large collections of text data (e.g., books, articles, and web pages) publicly available by September 2021. In our dataset, 27 vulnerability entries were posted before that date; 49 Apps have dependencies on any of those vulnerable library versions; ChatGPT successfully generated 20 exploits, achieving 41% (20/49) success rate. Meanwhile, three vulnerability entries were created after the cutoff date; six Apps depend on those vulnerable library versions; ChatGPT successfully generated four exploits, achieving 67% (4/6) success rate. For the data before September 2021, overfitting may occur and our evaluation may overestimate ChatGPT's capability. However, given the lower success rate of ChatGPT for vulnerabilities known before the cutoff date (i.e., 41% vs. 67%) and considering the large volume of training data used by ChatGPT (i.e., 570 GB), the overfitting issue seems insignificant.

educated assumption

6 DISCUSSION

From our study, we obtained the following insights about the strengths and weaknesses of ChatGPT.

ChatGPT is always able to generate security tests, even though the test quality varies a lot. According to ChatGPT itself, "OpenAI did not use reinforcement learning with human feedback to train me. Instead, I was pre-trained using a combination of unsupervised and supervised learning techniques, such as language modeling, auto-encoding, and sequence prediction. My training involved processing massive amounts of text data from the internet, which allowed me to learn patterns and relationships between words and phrases." [27]. The pre-trained knowledge enables ChatGPT to always produce tests in our study, but does not guarantee the quality. To provide quality assurance, future work can explore two directions. First, fine-tune or further train ChatGPT specially for test generation using the dialogue data between humans [27], so that ChatGPT learns to output better tests. Second, integrate ChatGPT with automatic compilation and testing, so that compilation/testing errors get used as feedback to help ChatGPT iteratively refine test generation.

explains applicability

Could we use cursor?

ChatGPT requires for exemplar Lib tests to generate security exploits, although such test examples are not always available or helpful. As shown in Section 4.3, without any test example, ChatGPT could not generate any exploit; with the test examples provided, however, ChatGPT generated exploits for 44% of the cases (24/55). The observations imply two things. First, ChatGPT can be effectively applied to generate security tests that mimic hand-crafted tests and exploit known vulnerabilities. Second, future work can put more effort into (1) providing more initial test examples, and (2) better training ChatGPT so that it can fully leverage the domain knowledge embedded in test examples to produce more successful vulnerability exploits.

Better at mimicing than exploit

Compared with the design of program analysis-based tools, ChatGPT's design seems weak in inter-procedural analysis but stronger in code generation. Based on our experience, when a method-to-test M indirectly calls vulnerable API(s), ChatGPT barely succeeded in vulnerability exploitation. One possible reason is that ChatGPT cannot relate the program context in different methods via caller-callee relations. Our observation implies the future direction of combining program analysis techniques with ChatGPT in novel ways, so that (1) ChatGPT fulfills more exploits when the program context is complex or there is a long call chain between M and vulnerable APIs, or (2) program analysis-based tools (especially dynamic analysis tools) obtain better results when ChatGPT-generated tests trigger the execution of more paths.

7 RELATED WORK

The related work includes detection of vulnerable dependencies, detection of vulnerable API usage, security test generation, and empirical studies on ChatGPT.

7.1 Detection of Vulnerable Dependencies

People built a variety of tools to detect vulnerable dependencies in software projects [1, 2, 10, 34, 41, 42, 47, 80]. For instance, npm-audit [34], snyk-test [47], AuditJS [42], RetireJS [41], and gammaray [1] scan JS applications for their package/library dependencies, compare those packages as well as versions against the known package versions in vulnerability databases (e.g., NVD [35]), and report a vulnerability for each found match. OWASP Dependency-Check [2] and SwiftDependencyChecker [80] implement the same technique, to separately reveal vulnerable dependencies in Java and Swift programs. However, recent studies and articles show that developers did not trust many of the vulnerabilities reported by these tools [4, 9, 66, 76]. For example, some reported vulnerabilities are not exploitable, as they exist in development-only or test dependencies and never get deployed as parts of released software [66, 69, 75]. Developers would like to see how vulnerabilities can be exploited, before fixing the reported vulnerabilities [4, 66]. Our research was motivated by developers' dissatisfaction with existing dependency checkers.

7.2 Detection of Vulnerable API Usage

To tackle the imprecision of coarse-grained version matching by existing dependency checkers, researchers and engineer created tools to detect invocation of vulnerable APIs or vulnerable usage of APIs [3, 5, 8, 58, 60, 68, 77, 79, 81, 86, 89]. Specifically, FindSecBug [3], SonarQube [5], Xanitizer [8], CogniCrypt [68], CryptoGuard [79], CryptoTutor [81], and SEADER [89] conduct program static analysis, to check whether Java cryptographic APIs are called with insecure parameter values or in wrong sequential orders. Fischer et al. [60] and Xu et al. [86] separately created machine learning-based tools to detect vulnerable API usage. Eclipse Steady [77] statically analyzes the security fixes (i.e., code changes) applied to a reported vulnerable library version, to locate the vulnerable code or APIs. It then combines static with dynamic analysis, to decide whether a given client app calls the vulnerable API(s) or reaches the vulnerable code through call path(s).

Although current tools can effectively locate vulnerable API usage by client apps, a recent study [88] shows that most developers refused to seriously consider the outputs by such tools or modify their API usage accordingly. One reason developers mentioned is that they wanted to see security exploit or the actual security impact, before addressing any reported vulnerability. Our study complements existing work, as it explores ways of generating security exploits to help persuade developers into addressing the vulnerabilities reported by existing detectors.

7.3 Security Test Generation

Various tools were built to generate security tests [12, 37, 54–57, 59, 62, 63, 70, 71, 83, 85]. Specifically, Marback et al. [70] and Xu et al. [85] created approaches, 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 those paths to executable test code via tool automation or manual effort. However, these approaches require lots of manual effort and domain knowledge. If a user is unable to precisely model all potential threats/attacks, or to accurately convert attack paths to executable tests, these approaches are ineffective in creating tests.

Traditional verification takes in a program and a specification of safety, and verifies whether the program satisfies the safety specification. Automatic exploit generation (AEG) [55–57, 62] twists program verification, by replacing typical safety properties with an exploitability property, and the verification process becomes finding a program path where the exploitability property holds. For instance, Ganapathy et al. [62] explores API-level exploitability with bounded model checking (BMC). AEG often suffers from scalability challenges (e.g., path explosion and the NP-hardness of

Produced with a Trial Version of PDF Annotator - www.PDFAnnotator.com

solving SMT queries in general). People also proposed fuzzing (i.e., fuzz testing) tools to generate security tests [12, 37, 59, 63, 83]. Fuzzing injects invalid, malformed, or unexpected inputs into a system to reveal software defects and vulnerabilities [50]. However, fuzzing is unable to explore deep paths; an inefficient initial seed can incur high runtime-overheads, because the mutants-to-generate depend on that seed. To overcome the limitations of both program verification and fuzzing, Alshmrany et al. [54] and Metta et al. [71] combined BMC with fuzzers to generate security tests.

Our research is different from all work mentioned above in two aspects. First, it adopts ChatGPT to generate test cases. Second, it mimics the exemplar test for a vulnerable library, to similarly generate a security test for client Apps built on top of that library.

7.4 Empirical Studies on ChatGPT

A few studies were recently done to assess ChatGPT's capability in programming or assisting programmers [65, 72, 74, 82, 84]. For instance, Nascimento et al. [72] chose four LeetCode questions to create prompts for ChatGPT. Jalil et al. [65] checked how well ChatGPT performs when answering the common questions in a popular software testing curriculum. Sobania et al. [82] evaluated ChatGPT on the standard bug fixing benchmark set—QuixBugs; they found it to fix 31 out of 40 bugs, outperforming the state-of-the-art. Tian et al. [84] assessed ChatGPT's capability in code generation, program repair, and code summarization. Nikolaidis et al. [74] evaluated ChatGPT and Copilot using LeetCode problems. Our study complements all research work mentioned above, because we applied ChatGPT to perform a totally different task: exploit generation.

8 CONCLUSION

We explored to use ChatGPT in generating security exploits, to demonstrate how vulnerabilities are propagated from software libraries to client applications. Our hypothesis was that ChatGPT could not generate tests effectively, because it may not have sufficient domain knowledge about security vulnerabilities, security exploitation, or the program context of client projects, neither does it seem to be capable of conducting complex program analysis. Surprisingly, we observed ChatGPT to work effectively in generating security tests, given prompts that cover the relevant domain knowledge. ChatGPT even outperformed state-of-the-art tools that leverage complex program analysis and genetic programming to generate diverse tests. This implies that in the future, we can create ChatGPT-based tools to automatically synthesize prompts and generate security tests. Researchers can also combine ChatGPT with program analysis techniques to generate higher-quality tests.

Our exploration of using distinct prompt templates shows considerably different results of ChatGPT, when the tool is given different types of prompts. The results provide two insights. First, ChatGPT is very sensitive to prompt quality; there is a huge gap among its outcomes when it is given good or bad prompts. Namely, to fully leverage the strengths of ChatGPT, we need to carefully design prompt templates. Second, the more domain knowledge included into the prompts, the better ChatGPT works to generate tests as we need.

Although some of the generated tests in our study are not effective in leveraging the known vulnerabilities, they surprisingly revealed some software bugs or vulnerabilities we did not initially anticipate to identify. We created four CVE entries for these bugs, and one of them has been confirmed by developers. This implies that ChatGPT is also promising in generating tests to reveal new software bugs or vulnerabilities. In the future, we also plan to integrate ChatGPT with existing test generation tools, to help developers detect and fix bugs more efficiently.

REFERENCES

[1] 2019. GitHub - nearform / gammaray: Node.js vulnerability scanner. https://github.com/nearform/gammaray.
[2] 2020. OWASP Dependency-Check. https://owasp.org/www-project-dependency-check/.

[3] 2021. Find Security Bugs. https://find-sec-bugs.github.io/

[4] 2021. npm audit: Broken by Design. https://overreacted.io/npm-audit-broken-by-design/.

[5] 2021. SonarQube. https://github.com/SonarSource/sonarqube.

[6] 2021. 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.

[7] 2021. 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.

[8] 2021. Xanitizer by RIGS IT - Because Security Matters. https://www.rigs-it.com/xanitizer/

[9] 2022. False Positives in Vulnerability Scanning: Why We Think We Can Do Better. https://www.lunasec.io/docs/blog/the-issue-with-vuln-scanners/.

[10] 2023. About Dependabot alerts. https://docs.github.com/en/code-security/dependabot/dependabot-alerts/about-dependabot-alerts.

[11] 2023. alibaba / fastjson. https://github.com/alibaba/fastjson.

[12] 2023. american fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[13] 2023. apache / commons-io. https://github.com/apache/commons-io.

[14] 2023. apache / cxf. https://github.com/apache/cxf.

[15] 2023. apache / httpcomponents-client. https://github.com/apache/httpcomponents-client.

[16] 2023. Apache Tika. https://tika.apache.org.

[17] 2023. Automatic fixing with snyk fix - Snyk User Docs. https://docs.snyk.io/snyk-cli/test-for-vulnerabilities/automatic-remediation-with-snyk-fix.

[18] 2023. Codec. https://commons.apache.org/proper/commons-codec/.

[19] 2023. Commons Compress - Overview. https://commons.apache.org/proper/commons-compress/.

[20] 2023. CVE-2020-28052 Detail. https://nvd.nist.gov/vuln/detail/cve-2020-28052.

[21] 2023. Dom4j. https://dom4j.github.io.

[22] 2023. eclipse / rdf4j. https://github.com/eclipse/rdf4j.

[23] 2023. FasterXML / jackson-databind. https://github.com/FasterXML/jackson-databind.

[24] 2023. FasterXML / jackson-dataformats-binary. https://github.com/FasterXML/jackson-dataformats-binary.

[25] 2023. FasterXML / jackson-modules-java8. https://github.com/FasterXML/jackson-modules-java8.

[26] 2023. haraldk / TwelveMonkeys. https://github.com/haraldk/TwelveMonkeys.

[27] 2023. How does ChatGPT actually work? https://www.zdnet.com/article/how-does-chatgpt-work/.

[28] 2023. Inside ChatGPT's Brain: Large Language Models. https://serokell.io/blog/language-models-behind-chatgpt.

[29] 2023. Introducing ChatGPT. https://openai.com/blog/chatgpt.

[30] 2023. junrar / junrar. https://github.com/junrar/junrar.

[31] 2023. Mockito. https://site.mockito.org/. Accessed on June 12, 2023.

[32] 2023. netplex / json-smart-v1. https://github.com/netplex/json-smart-v1.

[33] 2023. netplex / json-smart-v2. https://github.com/netplex/json-smart-v2.

[34] 2023. npm-audit. https://docs.npmjs.com/cli/v9/commands/npm-audit.

[35] 2023. NVD. https://nvd.nist.gov.

[36] 2023. OpenRefine. https://github.com/OpenRefine/OpenRefine.

[37] 2023. OSS-Fuzz. https://google.github.io/oss-fuzz/.

[38] 2023. OWASP / json-sanitizer. https://github.com/OWASP/json-sanitizer.

[39] 2023. OWASP Top Ten. https://owasp.org/www-project-top-ten/.

[40] 2023. Plexus Archiver Component. https://codehaus-plexus.github.io/plexus-archiver/index.html.

[41] 2023. Retire.js. https://retirejs.github.io/retire.js/.

[42] 2023. sonatype-nexus-community / auditjs: Audits an NPM package.json file to identify known vulnerabilities. https://github.com/sonatype-nexus-community/auditjs.

[43] 2023. spring-projects / spring-data-commons. https://github.com/spring-projects/spring-data-commons.

[44] 2023. spring-projects / spring-security. https://github.com/spring-projects/spring-security.

[45] 2023. srikanth-lingala / zip4j. https://github.com/srikanth-lingala/zip4j.

[46] 2023. stleary / JSON-java. https://github.com/stleary/JSON-java.

[47] 2023. Test - Snyk User Docs. https://docs.snyk.io/snyk-cli/commands/test.

[48] 2023. The Legion of the Bouncy Castle. https://www.bouncycastle.org.

[49] 2023. The Next Supply Chain Attack Vector: Open-Source Software. https://www.supplychainbrain.com/blogs/1-think-tank/post/36830-the-next-supply-attack-vector-open-source-software.

[50] 2023. What Is Fuzz Testing and How Does It Work? | Synopsys. https://www.synopsys.com/glossary/what-is-fuzz-testing.html.

[51] 2023. xerial / snappy-java. https://github.com/xerial/snappy-java.

[52] 2023. XStream. https://x-stream.github.io.

[53] 2023. ZT Zip. https://github.com/zeroturnaround/zt-zip.

[54] Kaled M. Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, and Lucas C. Cordeiro. 2021. FuSeBMC: An Energy-Efficient Test Generator for Finding Security Vulnerabilities in C Programs. In *Tests and Proofs*, Frédéric Loulergue and Franz Wotawa (Eds.). Springer International Publishing, Cham, 85–105.

[55] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. *Commun. ACM* 57, 2 (2014), 74–84.

[56] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. 143–157. https://doi.org/10.1109/SP.2008.17

[57] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *2012 IEEE Symposium on Security and Privacy*. 380–394. https://doi.org/10.1109/SP.2012.31

[58] Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. 2021. Code-Based Vulnerability Detection in Node.Js Applications: How Far Are We?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1199–1203. https://doi.org/10.1145/3324884.3421838

[59] Jared Demott, Dr Richard, R.J. Enbody, Dr William, and William Punch. 2007. Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing. In *Black Hat and DEFCON*.

[60] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 121–136.

[61] Gordon Fraser and Andrea Arcuri. 2011. 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*. 416–419.

[62] Vinod Ganapathy, Sanjit A. Seshia, Somesh Jha, Thomas W. Reps, and Randal E. Bryant. 2005. Automatic Discovery of API-Level Exploits. In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, USA) *(ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 312–321. https://doi.org/10.1145/1062455.1062518

[63] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact at Microsoft. *Queue* 10, 1 (jan 2012), 20–27. https://doi.org/10.1145/2090147.2094081

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

[65] S. Jalil, S. Rafi, T. D. LaToza, K. Moran, and W. Lam. 2023. ChatGPT and Software Testing Education: Promises & Perils. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE Computer Society, Los Alamitos, CA, USA, 4130–4137. https://doi.org/10.1109/ICSTW58534.2023.00078

[66] Md Mahir Asef Kabir, Ying Wang, Danfeng Yao, and Na Meng. 2022. How Do Developers Follow Security-Relevant Best Practices When Using NPM Packages?. In *2022 IEEE Secure Development Conference (SecDev)*. IEEE Computer Society, Los Alamitos, CA, USA, 77–83. https://doi.org/10.1109/SecDev53368.2022.00027

[67] Hong Jin Kang, Truong Giang Nguyen, Bach Le, Corina S Păsăreanu, and David Lo. 2022. Test mimicry to assess the exploitability of library vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 276–288.

[68] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, et al. 2017. CogniCrypt: supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 931–936.

[69] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering* 23 (2018), 384–417.

[70] Aaron Marback, Hyunsook Do, Ke He, Samuel Kondamarri, and Dianxiang Xu. 2009. Security test generation using threat trees. In *2009 ICSE Workshop on Automation of Software Test*. 62–69. https://doi.org/10.1109/IWAST.2009.5069042

[71] Ravindra Metta, Raveendra Kumar Medicherla, and Samarjit Chakraborty. 2022. BMC+Fuzz: Efficient and Effective Test Generation. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1419–1424. https://doi.org/10.23919/DATE54114.2022.9774672

[72] Nathalia Nascimento, Paulo Alencar, and Donald Cowan. 2023. Comparing Software Developers with ChatGPT: An Empirical Investigation. arXiv:2305.11837 [cs.SE]

[73] Duc Cuong Nguyen, Erik Derr, Michael Backes, and Sven Bugiel. 2020. Up2Dep: Android Tool Support to Fix Insecure Code Dependencies. In *Annual Computer Security Applications Conference* (Austin, USA) *(ACSAC '20)*. Association for Computing Machinery, New York, NY, USA, 263–276. https://doi.org/10.1145/3427228.3427658

[74] Nikolaos Nikolaidis, Karolos Flamos, Daniel Feitosa, Alexander Chatzigeorgiou, and Apostolos Ampatzoglou. [n. d.]. 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.

[75] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable Open Source Dependencies: Counting Those That Matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Oulu, Finland) *(ESEM '18)*. Association for Computing Machinery, New York, NY, USA, Article 42, 10 pages. https://doi.org/10.1145/3239235.3268920

[76] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2020. Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering* 48, 5 (2020), 1592–1609.

[77] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2020. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering* 25, 5 (2020), 3175–3215.

[78] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 383–387. https://doi.org/10.1109/MSR.2019.00064

[79] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng Yao. 2019. 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*. 2455–2472.

[80] Kristiina Rahkema and Dietmar Pfahl. 2022. 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* (Pittsburgh, Pennsylvania) *(MOBILESoft '22)*. Association for Computing Machinery, New York, NY, USA, 107–111. https://doi.org/10.1145/3524613.3527806

[81] Larry Singleton, Rui Zhao, Myoungkyu Song, and Harvey Siy. 2020. CryptoTutor: Teaching Secure Coding Practices through Misuse Pattern Detection. In *Proceedings of the 21st Annual Conference on Information Technology Education*. 403–408.

[82] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. arXiv:2301.08653 [cs.SE]

[83] Ari Takanen, Jared Demott, Charles Miller, and Atte Kettunen. 2017. *Fuzzing for Software Security Testing and Quality Assurance, Second Edition*. Artech House.

[84] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F. Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant – How far is it? arXiv:2304.11938 [cs.SE]

[85] Dianxiang Xu, Manghui Tu, Michael Sanford, Lijo Thomas, Daniel Woodraska, and Weifeng Xu. 2012. Automated Security Test Generation with Formal Threat Models. *IEEE Transactions on Dependable and Secure Computing* 9, 4 (2012), 526–540. https://doi.org/10.1109/TDSC.2012.24

[86] Zhiwu Xu, Xiongya Hu, Yida Tao, and Shengchao Qin. 2020. Analyzing Cryptographic API Usages for Android Applications Using HMM and N-Gram. In *2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, 153–160.

[87] Sebastian Zander, Grenville Armitage, and Philip Branch. 2007. A survey of covert channels and countermeasures in computer network protocols. *IEEE Communications Surveys & Tutorials* 9, 3 (2007), 44–57. https://doi.org/10.1109/COMST.2007.4317620

[88] Ying Zhang, Md Mahir Asef Kabir, Ya Xiao, Danfeng Yao, and Na Meng. 2022. Automatic Detection of Java Cryptographic API Misuses: Are We There Yet? *IEEE Transactions on Software Engineering* 49, 1 (2022), 288–303.

[89] Ying Zhang, Ya Xiao, Md Mahir Asef Kabir, Danfeng (Daphne) Yao, and Na Meng. 2022. Example-Based Vulnerability Detection and Repair in Java Code. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension* (Virtual Event) *(ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 190–201. https://doi.org/10.1145/3524610.3527895