SoftSEC
Institute of Software Security

TUHH
Hamburg University of Technology

(Dynamic) Security Testing and Exploit Generation

Emanuele Iannone

Institute of Software Security, TU Hamburg, Germany

International School

University of Salerno, Avellino, 16 June 2025

Whoami

Emanuele Iannone

em***.***ne@tuhh.de

emaiannone.github.io

@EmanueleIannon3

Nationality Italian

Career

B.Sc. in Computer Science @ UNISA — 2015 - 2018
Refactoring for Android Energy Consumption

M.Sc. in Computer Science @ UNISA — 2018 - 2020
Test Generation for 3rd-party Vulnerabilities

Ph.D. in Computer Science @ UNISA — 2020 - 2024
Empirical Comprehension and Automated Approaches for Software Vulnerabilities

Postdoctoral Researcher @ TU Hamburg — 2023 - curr.
Sec4AI4Sec EU Horizon Project (Grant ID: 101120393)

Research Interests

Security Vulnerability Testing

Mining Software Repositories & Software Analytics (for security)

AI for Software Security Engineering

Security Testing… uh?

What is software testing?

Security Testing… uh?

What is software testing?

The key activity for finding defects in a test item.[1]

A **test item** can be the whole system, a subsystem, a class, a method, a function, etc.

If a test item can be executed, software testing can be dynamic, i.e., the source code is executed through **test cases**.

A **test case** runs the test item with **specific input** and observes whether the results obtained match the **expected** ones, given certain **preconditions**.

The expected results are decided by a source of information called **oracle**, which can often (but not always) be the **specification** of the test item.

[1] *ISO/IEC/IEEE International Standard - Software and systems engineering --Software testing --Part 1:General concepts (ISO/IEC/IEEE 29119-1:2022)*

Security Testing… uh?

What is security testing?

Security Testing… uh?

What is security testing?

An activity that verifies the protection against unauthorized access and allows entry for authorized parties.[1]

Access = Use, read, modify. A test item can also be data, not just code!

When no context is given, software testing aims to find **functional defects** (bugs).

However, software testing can also find other kinds of defects, including security defects (i.e., **vulnerabilities**) that threaten some **security properties** of the test item.

Our scope: Vulnerabilities affecting the source code directly.

- Ignore those affecting the whole architecture, the infrastructure, or the network.

[1] *ISO/IEC/IEEE International Standard - Software and systems engineering --Software testing --Part 1:General concepts (ISO/IEC/IEEE 29119-1:2022)*

Static Testing vs. Dynamic Testing

Historically: Testing is a **dynamic activity**. Period.

Today: Testing can be static or dynamic.

It does not really matter as long as we agree on the terminology.

Our focus is on dynamic testing, but let's examine the key differences between it and static testing (commonly called “static analysis”).

Static Testing

- No code is run; only inspected (less realistic).
- Requires the source code.
- Multiple code paths can be explored.
- They can prove the absence of a vulnerability but cannot prove their presence.

Dynamic Testing

- The code is actually run (more realistic).
- Requires an interface to run the code.
- Only some code paths can be explored.
- They can prove the presence of a vulnerability but cannot prove their absence.

(Dynamic) Security Testing: Pros and Cons

- ⊕ Less subject to false positives: The actual reaction of the test item is more reliable than a “warning” from a static analysis tool.
- ⊕ The findings can be proof-of-concepts of vulnerabilities (good for disclosing them).
- ⊕ They can reveal issues caused by the configuration and the host (beyond code).
- ⊖ It requires buildable AND runnable code (not good for early development stages).
- ⊖ High expertise might be required:
  - Tool designers must think of many scenarios to cover.
  - Tool users must learn to configure the tool adequately.
- ⊖ Difficult to triangulate the exact location of the vulnerabilities found.
- ⊖ Cannot find “*weak code smell*”, but only exploitable issues.

(Dynamic) Security Testing Techniques

Vulnerability Scanning

Penetration Testing

Fuzzing

Code-level Security Testing

We'll briefly see each of them, focusing more on the code-level security testing.

Vulnerability Scanning

Vulnerability Scanners

Vulnerability scanning consists of systematically stimulating an application with precise inputs (i.e., **payloads**) to uncover vulnerabilities.

- Very often, the application is web-based (but not necessarily).
- The inputs tested are pre-defined within the vulnerability scanner because they historically/empirically demonstrated to trigger **specific vulnerability types** effectively (users can add custom inputs).

Scanning is often called **D**ynamic **A**pplication **S**ecurity **T**esting (DAST).

- People often (mis)use this term, including fuzzing and other techniques within it.
- No precise/standard definition of what DAST really is…

Vulnerability Scanners: The Process

In essence, scanners see the tested application as a black box: The inputs are sent from the application interface, e.g., *HTTP requests*.

- **White-box scanners** exist but are less common.

Steps of a standard (web) scanner

1) Discovery. The scanner runs *crawlers* (a.k.a. *spiders)* to map the URL/endpoints that can accept the inputs. Crawlers can follow static and/or dynamically generated links.

2) Send Input. For each vulnerability type supported, the scanner starts sending the pre-defined inputs to the entry points mapped.

3) Interpret Response. Assess whether an input "triggered" a **behavior that indicates the presence of a vulnerability** (example in the next slide).

4) Report. Document the findings obtained.

Vulnerability Scanners: An Example

Let us suppose the crawler found a **web page with a search form** (one text field and a submit button).

What input would you try?

Vulnerability Scanners: An Example

Let us suppose the crawler found a **web page with a search form** (one text field and a submit button).

What input would you try?

For Cross-site Scripting (XSS): Send an input like `<script>alert(VALUE);</script>`.

- If the browser shows a popup containing VALUE, then the application is likely vulnerable to a (reflected) XSS.

For Buffer Overflow: Send a VERY long input.

- If the server stops responding or the session is abruptly closed, then a crash has likely occurred due to a buffer overflow.

Some scanners can find likely vulnerabilities by simply looking at the responses obtained during the crawling without sending any malicious input (passive checks).

- For example, HTTP responses containing cookies without the “HttpOnly” flag.

Vulnerability Scanners: Pros and Cons

- ➕ Good for beginners (rely on the pre-defined checks).
- ➕ Support many common vulnerability types (SQLi, XSS, Buffer Overflow, Input Validation, Improper Access Control, etc.)
- ➕ Fully automated and somewhat easy to plug into a Continuous Integration pipeline.
- ➖ The application must be **built AND deployed** (so it can only be done at later development stages)
- ➖ More **advanced checks** must be defined by the user, which is not easy for beginners.
- ➖ Only a few are free and open-source (e.g., *OWASP ZAP*); **most are commercial** (though there can be *free community editions*).

Fuzzing

Fuzzing: The Origin

In 1988, Barton Miller (University of Wisconsin) asked his students to write a small program that *"fuzzes"* various UNIX utilities (programs that can be called from the command line) with "random" inputs to see what happened.

This helped reveal several inputs that led them to crash or hang.

What's the relation with security?

Fuzzing: The Origin

In 1988, Barton Miller (University of Wisconsin) asked his students to write a small program that *"fuzzes"* various UNIX utilities (programs that can be called from the command line) with "random" inputs to see what happened.

This helped reveal several inputs that led them to crash or hang.

What's the relation with security?

Crashes and hangs violate the security property of availability.

- Think of a server application that stops serving clients…

They are the consequence of various **bad things** (partial list):

- Segmentation fault (e.g., due to buffer overrun).
- Illegal memory access (Use-after-free, double-free, etc.).
- Resource exhaustion.
- Unhandled errors/exceptions.
- Infinite loop.

Fuzzing: Is Random the Right Way?

With random inputs, something will come out… sooner or later. But how much “later”?

What’s the problem with full randomness?

Fuzzing: Is Random the Right Way?

With random inputs, something will come out… sooner or later. But how much “later”?

What’s the problem with full randomness?

Unfortunately, going fully random is not:

- **Effective**: Most interesting bugs hide behind *corner cases* (out-of-the-ordinary input), often in program paths that are difficult to reach.
  - Indeed, the “defect distribution” in the input space is not uniform.
- **Efficient**: A random (uniform) search usually leads to input that does not reveal any bugs, wasting precious time (*budget*).

Fuzzing: Guidance Strategies

For this, researchers added some guidance strategies to the input generation. A guidance criterion is good when:

- It covers common corner cases, such as empty strings, very long strings, min/max values of integer ranges, nulls, zeros, and negative values. (**Boundary Value Analysis**).
  - Rationale: the majority of bugs hide behind them.
- It is aware of the consumer of the input. For example:
  - If the input is a string that ends up in a `printf()` → format string characters (%s, %x, %n)
  - If the input is a string that ends up in a web page → HTML tags
  - If the input is a string that ends up in an SQL statement → SQL keywords
  - If the input is a string that ends up in a file path → Slashes and dots
  - etc.

Families of Fuzzing

Several families exist. These are the most prominent (not exhaustive).

Blackbox Fuzzing

Greybox Fuzzing

Whitebox Fuzzing

Mutation-based Fuzzing

Grammar-based Fuzzing

Blackbox Fuzzing

The tested program is only called from the “outside”, i.e., through its interface. No source code is needed (like the original approach by Miller).

Mutation-based Fuzzing

The user provides an initial input called seed. The seed can come from data observed during the program’s ordinary use. Multiple seeds can be used, forming a **seed corpus**.

The fuzzer picks one of the seeds and applies random mutations according to specific rules (chosen by the fuzzer’s designer) to generate new inputs to test.

Very easy to design, but the success **highly depends on the seed corpus**.

- Selecting a well-diversified seed corpus is crucial (and challenging).

Mutation-based Fuzzing: A Minimal Working Example

```
def insert_random_character(s):
    """Returns s with a random character inserted"""
    pos = random.randint(0, len(s))
    random_character = chr(random.randrange(32, 127))
    return s[:pos] + random_character + s[pos:]

for i in range(10):
    print(repr(insert_random_character('A quick brown fox')))
```

```
>>> 'A quick brvown fox'
>>> 'A quwick brown fox'
>>> 'A qBuick brown fox'
>>> 'A quick broSwn fox'
>>> 'A quick brown fvox'
```

More mutation operators can be defined.

If more operators exist, a randomness factor can be used to select which to call.

The seed can be randomly picked from a corpus rather than being a specific one.

More on https://www.fuzzingbook.org/html/MutationFuzzer.html

Blackbox Fuzzing

The tested program is only called from the “outside”, i.e., through its interface. No source code is needed (like the original one by Miller).

Grammar-based Fuzzing

The user provides a set of **specifications of the legitimate inputs** that the tested program can accept. Such specifications are often implemented as grammars.

- A grammar describes the *syntax* of an input.

The fuzzer generates “random” input data using the grammar.

- The input will surely be legal (no boundary value analysis) but can explore deeper and hard-to-reach program paths.
- Adding a mutational component on top can add the boundary value analysis.

Easy to design, but the challenge stands in providing the correct grammar for each input the program can accept (there can be many).

Grammar-based Fuzzing: Example Grammar

Grammar for arithmetic expressions

```
<start>   ::= <expr>
<expr>    ::= <term> + <expr> | <term> - <expr> | <term>
<term>    ::= <term> * <factor> | <term> / <factor> | <factor>
<factor>  ::= +<factor> | -<factor> | (<expr>) | <integer> | <integer>.<integer>
<integer> ::= <digit><integer> | <digit>
<digit>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

A (context-free) grammar consists of a `<start>` symbol and a set of expansion rules that indicate how the `<start>` symbol and other nonterminal symbols can be expanded (until all are exhausted). The vertical bar indicates alternative expansions we can take.

Let us generate a random input as if we were a fuzzer. We begin with `<start>` and randomly pick expansion rules every time we have a nonterminal symbol.

`<start>` □ `<expr>` □ `<term> + <expr>` □ `<factor> + <expr>` □ `-<factor> + <expr>` □ `-<integer> + <expr>` □ `-<digit> + <expr>` □ `-6 + <expr>` □ [continue on your own]

More on: https://www.fuzzingbook.org/html/Grammars.html

Greybox Fuzzing

Similar to a mutation-based fuzzer with a seed corpus, but it also observes the program execution to try to understand the “promising” inputs.

A seed is picked from the corpus. If one of the mutations covers new branches or paths, that seed is rewarded as a “progressive”, i.e., it will be selected more often than others.

- To measure the coverage, access to the source code is needed.
- Since it only needs to map the branches and paths covered, lightweight instrumentation is sufficient.
  - This is why it is called “gray” rather than “white”.

The reward model and how to spend the time budget are decided by the so-called power schedule (many algorithms exist).

Greybox Fuzzing: Notable Tools

Google's American Fuzzy Lop (AFL) is perhaps the most famous.

- It has been used to discover many vulnerabilities that often end up being disclosed as CVEs.

Many extensions (called flavors):

- AFLGo, AFLFast, AFLNet, AFLSmart, ...
- Each aims to improve the various components in it, mainly the power schedule and the seed selection mechanism.

Available at: https://github.com/google/AFL/

More on https://www.fuzzingbook.org/html/GreyboxFuzzer.html

Whitebox Fuzzing

It aims to reach specific program paths and generates input covering them. This requires deeper program analysis (not just instrumentation).

- Often use **Control-** and **Data-flow analysis**.
- Many are based on **symbolic execution** (or its variants) and **constraint satisfaction**.
- **Meta-heuristic search algorithms** are also used.
- Takes significant time.

Greybox vs. Whitebox: Greybox aims to discover new execution paths with random mutations, while whitebox targets specific program paths.

Example: Microsoft SAGE (academic tool).

Fuzzing: Pros and Cons

- ⊕ Simple to set up (especially blackbox ones).
- ⊕ Fully automated and somehow easy to plug into a Continuous Integration pipeline.
- ⊕ Produces a few false positives (crashes are self-explanatory).
- ⊖ Requires a significant time to return many findings.
- ⊖ Not all findings are vulnerabilities: Need an effective oracle (humans).
- ⊖ Existing fuzzers mainly target program binaries.
  - However, they can work with anything if designed properly, e.g., APIs and GUIs.

Penetration Testing

Penetration Testing (Pen Testing)

Authorized, simulated cyber attacks against a computer system (application, OS, network) to identify exploitable vulnerabilities.

- Simulated = the goal is to find security problems, not to cause real harm.

Typically, it is performed by the so-called red team, which can be made of sysadmins (internal) or trusted security consultants.

- It can also be done in a “competition” style with a blue team (with bounty).

The tested system is often not in actual production but in a controlled test environment (though realistic).

Phases of Penetration Testing

1. Reconnaissance and Infomation Gathering
   - Collect as much information as possible on the target (whois, nslookup).
2. Target Scanning
   - Obtain information about all the available hosts within the network.
   - Map the attack surface (port scanning, fingerprinting, web crawling).
3. Vulnerability Assessment and Exploitation
   - Check the **exploitability** of vulnerabilities (start with known vulnerabilities first).
   - Try to achieve the *maximum harm*, i.e., gain root privileges (e.g., spawn a root shell or retrieve the credentials of an admin account).
4. Post-Exploitation and Reporting
   - Remove all traces and document the process to the client.

The Reality of Penetration Testing

Pen testing is **NOT a technique**; it's an activity driven by people.

- Very expensive.
- The success depends on the expertise of the testers.

Its activities are often supported by existing automated tools (almost off the shelf):

- Vulnerability scanners (Burp Suite, ZAP, …)
- Metasploit (the essential toolkit for pen testers).

At the same time, they often require extensive **manual hacking** (few people are skilled enough to offer these services) and **custom scripts**.

Code-level Security Testing

Code-level Security Testing

Idea: Write and run specific test cases targeting a code component (e.g., a class method) to check if *"it works from a security perspective"*.

Key differences with "ordinary" functional testing:

- Focus on the security specification defined on the tested component.
- Focus on the security risks affecting the tested component.

We can also invoke 2+ components together.

- Useful when the real target component cannot be invoked easily (e.g., *private*).

Let's quickly go over code-level "ordinary" software testing.

Software Testing 101

Software testing is the default activity to find functional defects/bugs by running code (dynamic).

- **Defects/bugs** are caused by a **discrepancy** between the implementation of the component and its functional specification (dictating its expected behavior).
- Specification: A more or less formal description explaining how the component is expected to behave at certain inputs and preconditions.

A **test case** runs the component with **one input** and observes whether the results obtained match the **expected** ones (given the precondition).

- Often relying on automated assertions that fail when there is a discrepancy.
- Test cases are often implemented as code and run automatically via **testing frameworks**, e.g., JUnit, Pytest, Selenium, and many others.

Software Testing 101: Example of a Test Cases

```
public class NumberUtils {
    public static boolean isEven(int num) {
        return num % 2 == 1;
    }
}
```

Buggy method!

What's wrong with this method?

Software Testing 101: Example of a Test Cases

```
public class NumberUtils {
    public static boolean isEven(int num) {
        return num % 2 == 1;
    }
}
```

Buggy method!

What tests would you write?

Software Testing 101: Example of a Test Cases

```java
public class NumberUtils {
    public static boolean isEven(int num) {
        return num % 2 == 1;
    }
}
```

Buggy method!

Setup: Prepare the input data and the state of the object to invoke. (Here, there is nothing to prepare; just send one integer).

Action: Invoke the target method.

Verification: Check whether the actual outcomes match the expected outcomes (through *assertions*).

```java
public class NumberUtilsTest {
    @Test
    public void testIsEvenWithEvenNumber() {
        assertTrue(NumberUtils.isEven(4));
    }

    @Test
    public void testIsEvenWithOddNumber() {
        assertFalse(NumberUtils.isEven(5));
    }
}
```

Software Testing 101

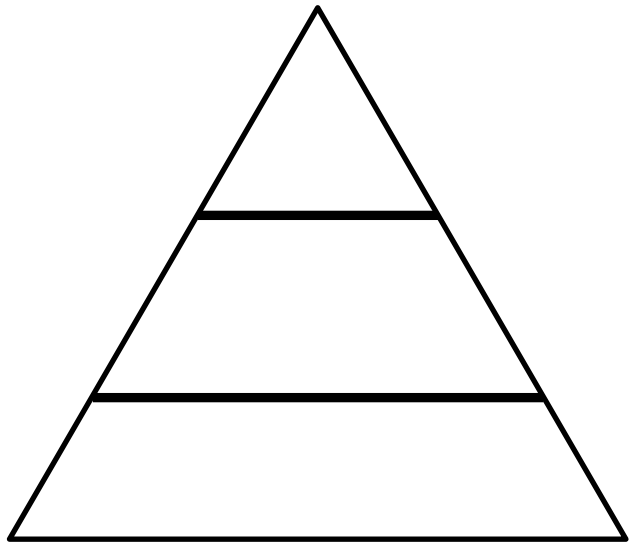
Theoretical goal: find all defects.

- Impossible, infinite execution domain.
- We cannot prove that a component is defect-free.

Practical goal: maximize the number of failures.

- Writing as many (non redundant) tests as possible.

Not just components! The test target depends on the granularity level.



System

Integration

Unit (Component)

More realistic (close to user interactions)
More complex
Find common bugs
Some needed

Less realistic (distant from user interactions)
Less complex
Find uncommon bugs
MANY needed

Software Testing for Code-level Security

Now, we want to reuse this knowledge to test the security of components.

Can the common security properties be a starting point?

- Based on the "CIA Triad" (*Confidentiality*, *Integrity*, *Availability*), extended with *Accountability*, *Non-Repudiation*, and *Authenticity*.
- Too high-level: They do not help select precise security test cases.

We resort to security specifications (requirements).

- Just like functional specification but focused on security (no surprise).
- Example: *"The method must encrypt its data before writing to a file."*

Security Requirements

Suppose we want to define a security requirement for a login method that (1) receives the input and password text fields from the HTTP (POST) request, (2) checks the database for the existence of the user and password match, (3) returns “True” if the user can be authorized, else “False”.

What security requirement would you define?

Security Requirements

Suppose we want to define a security requirement for a login method that (1) receives the input and password text fields from the HTTP (POST) request, (2) checks the database for the existence of the user and password match, (3) returns “True” if the user can be authorized, else “False”.

What security requirement would you define?

One possibility (not realistic, just an example):

- The login must detect failed attempts and block the IP if five failed attempts occurred within 1 minute.

Security specifications are functional descriptions of security mechanisms!

- For this, they are also known as security features.

Security Requirements

Now, given this “brute force prevention” mechanism for a login method…

What could be the right test cases?

Security Requirements

Now, given this “brute force prevention” mechanism for a login method…

What could be the right test cases?

Examples (not complete, many other cases exist):

- Fail the login **four** times within 1 minute. Expected: no auth, but no block.
- Fail the login **five** times within 1 minute. Expected: no auth, but no block.
- Fail the login **six** times within 1 minute. Expected: no auth, and sixth is blocked.
- Fail the login **four** times within 1 minute and **twice within the second minute**. Expected: no auth, but no block.

Take-home: As long as we have well-defined security requirements, we can select test cases just as ordinary functional test cases.

Security Requirements: The Reality

Unfortunately, not all gold that glitters. This approach works as long as:

- We identify all security requirements (**completeness**).
- Each requirement is well defined (**correctness**).

Possible solution

Rely on threat modeling, particularly on abuse-misuse cases, to identify as many security requirements as possible.

- **Problems**: time-consuming, expert-driven, lack of automation.

We need another approach...

Risk-based Security Testing

Instead of defining requirements, we select security test cases based on knowing which vulnerability types can affect the tested component.

- Catalogs of vulnerabilities: MITRE's CWE; OWASP Top 10 Threats.
- Catalogs of attacks: MITRE's CAPEC.

Not just boring catalogs: We can also rely on security testing guides:

- **OWASP WSTG** - Web Security Testing Guide (practical).
- **OWASP ASVS** - Application Security Verification Standard (high-level).
- **OWASP Cheat Sheet Series** (developer perspective more than tester).

So, a test case based on risks checks whether the tested component is vulnerable to certain attacks (e.g., because of improper protection).

Risk-based Security Testing

Suppose we want to find the security risks of the same login method as before.

What vulnerability types would you look at?

Risk-based Security Testing

Suppose we want to find the security risks of the same login method as before.

What vulnerability types would you look at?

Examples (not complete):

- CWE-89 – SQL Injection (if the method queries to an SQL database).
- CWE-79 – XSS (as username and password are free text fields).
- CWE-307 – Brute Force (just as the requirements defined before).
- CWE-387 – Session Fixation (it is recommended to have a new unpredictable session token generated after a successful login and never reuse old ones).

Risk-based Security Testing

Suppose we want to find the security risks of the same login method as before.

What vulnerability types would you look at?

Examples (not complete):

- CWE-89 – SQL Injection (if the method queries to an SQL database).
- CWE-79 – XSS (as username and password are free text fields).
- CWE-307 – Brute Force (just as the requirements defined before).
- CWE-387 – Session Fixation (it is recommended to have a new unpredictable session token generated after a successful login and never reuse old ones).

What test cases would you write for CWE-89?

Risk-based Security Testing

Suppose we want to find the security risks of the same login method as before.

What vulnerability types would you look at?

Examples (not complete):

- CWE-89 – SQL Injection (if the method queries to an SQL database).
- CWE-79 – XSS (as username and password are free text fields).
- CWE-307 – Brute Force (just as the requirements defined before).
- CWE-387 – Session Fixation (it is recommended to have a new unpredictable session token generated after a successful login and never reuse old ones).

What test cases would you write for CWE-89?

Input: `OR 1=1;--` Expected: login granted even with wrong credentials.

Security Test Case: An Example

CVE-2018-1274 in Spring Data Commons. The resolution of "property paths" (a notation to access nested object fields) in method `PropertyPath.create()` could use too many resources if deeply nested property paths were supplied (user-controlled).

```
private static PropertyPath create(String source,
                                   TypeInformation<?> type,
                                   String addTail,
                                   List<PropertyPath> base) {



  PropertyReferenceException exception = null;
  PropertyPath current = null;
  try {
      current = new PropertyPath(source, type, base);
      ...
  }
  ...
}
```

Vulnerable method

```
@Test
public void rejectsTooLongPath() {
    String source = "foo.bar";
    for (int i = 0; i < 9; i++) {
        source = source + "." + source;
    }
    assertThat(source.split("\\.").length, is(greaterThan(1000)));
    final String path = source;
    exception.expect(IllegalArgumentException.class);
    PropertyPath.from(path, Left.class);
}
```

`PropertyPath.from()` calls `PropertyPath.create()` indirectly!

Fails: There is a problem!

Security Test Case: An Example

CVE-2018-1274 in Spring Data Commons. The resolution of "property paths" (a notation to access nested object fields) in method `PropertyPath.create()` could use too many resources if deeply nested property paths were supplied (user-controlled).

```
private static PropertyPath create(String source,
                                   TypeInformation<?> type,
                                   String addTail,
                                   List<PropertyPath> base) {
    if (base.size() > 1000) {
      throw new IllegalArgumentException(PARSE_DEPTH_EXCEEDED);
    }
  PropertyReferenceException exception = null;
  PropertyPath current = null;
  try {
      current = new PropertyPath(source, type, base);
      ...
  }
  ...
}
```

Vulnerable method

Developer fix: max 1000 segments

```
@Test
public void rejectsTooLongPath() {
    String source = "foo.bar";
    for (int i = 0; i < 9; i++) {
        source = source + "." + source;
    }
    assertThat(source.split("\\.").length, is(greaterThan(1000)));
    final String path = source;
    exception.expect(IllegalArgumentException.class);
    PropertyPath.from(path, Left.class);
}
```

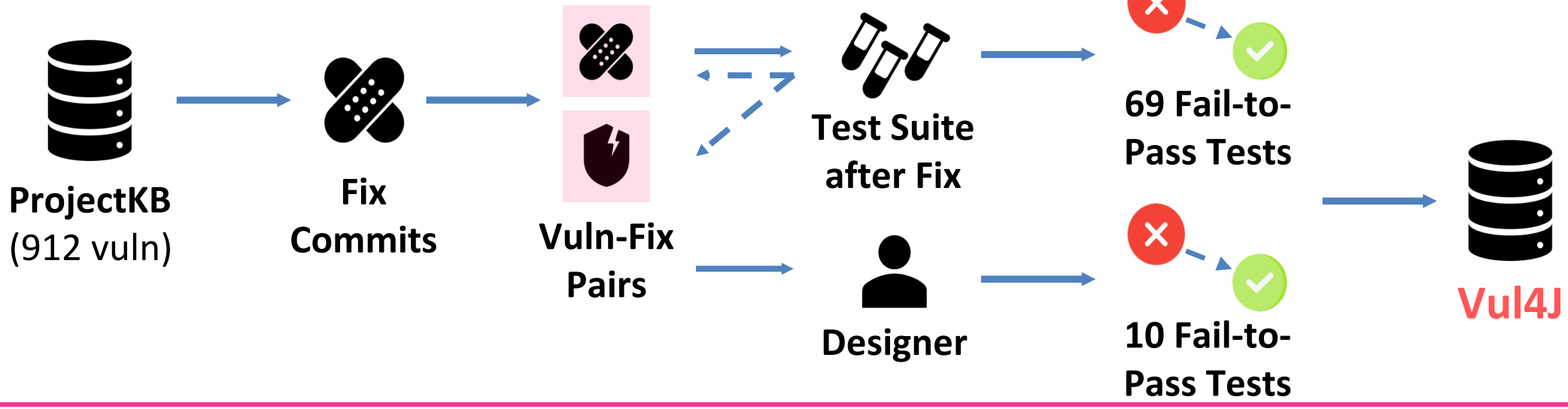
`PropertyPath.from()` calls `PropertyPath.create()` indirectly!

Passes: Fix looks good!

Vul4J: A Catalog of Security Test Cases

The previous test case is contained in Vul4J[1], a dataset of 79 known vulnerabilities with fix commits and test cases like the one seen earlier.

- Fail in the vulnerable version, pass in the fixed versions.
- Also called *Proof of Vulnerabilities* (PoV).



SEC 4AI4 SEC In project *Sec4AI4Sec*, we **further extended** this dataset with more tests, which were found and generated with AI. The extension is called Vul4J+.

[1]*Q. -C. Bui, R. Scandariato and N. E. D. Ferreyra, "Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques," 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), Pittsburgh, PA, USA, 2022.*

Code-level Security Testing: Pros and Cons

- ⊕ Closer to the developer: the tests are more focused and customizable.
  - Unlike other dynamic testing approaches.
- ⊕ Can be done as soon as the development starts.
  - No other dynamic testing approach can do this!
- ⊕ Test cases can be versioned and reused multiple times (especially in CI pipelines), also guarding against regressions.
- ⊖ Must be designed and implemented with attention (no “fire and forget”) and have to be maintained.
  - It seems security tests are an *afterthought, a*dded only to demonstrate a fix is correct (we need more empirical studies to confirm this behavior).

Code-level Security Testing: Developers' View

How do developers perceive unit testing for security?

- A survey of 31 open-source developers in 2017 confirmed that code-level security testing is seen as really effective but difficult to use.[1]
- An analysis of 525 StackOverflow posts in 2021 highlighted the developer pain points in writing unit and integration tests for security, mainly concerning the implementation rather than the design of tests.[2]
  - Example: **mocking** the external components correctly or **bypassing** authentication and authorization layers to test the protected components.

Opportunity: Automated generation tools could relieve this burden!

[1] *P. Morrison, B. H. Smith, and L. Williams. 2017. Surveying Security Practice Adherence in Software Development. In Proceedings of the Hot Topics in Science of Security: Symposium and Bootcamp (HoTSoS). https://doi.org/10.1145/3055305.3055312*

[2] *D. Gonzalez, P. Perez, and M. Mirakhorli. 2021. Barriers to Shift-Left Security: The Unique Pain Points of Writing Automated Tests Involving Security Controls. In Proceedings of the 15th ACM/IEEE Int. Symp. on Empirical Software Engineering and Measurement (ESEM '21). https://doi.org/10.1145/3475716.3475786*

Automated Code-level Security Testing (1/6)

Let's start with approaches working at the system level (through API).


Attack Patterns for Black-Box Security Testing of Multi-Party Web Applications

Sudhodanan et al.[1]: Targets *multi-party web applications* (like PayPal payment systems) by scanning the app with OWASP ZAP and then building test cases on top of its findings to verify the feasibility of certain attacks.


Automated Black-box Testing of Mass Assignment Vulnerabilities in RESTful APIs

Corradini et al.[2]: Targets any RESTful APIs, creating test cases that inject fields in server resources (through HTTPs) and verifying whether a read-only field is overwritten (confirming the presence of a *mass assignment vulnerability*).

[1] *A. Sudhodanan et al. "Attack Patterns for Black-Box Security Testing of Multi-Party Web Applications". NDSS 2016.*
[2] *D. Corradini et al. "Automated Black-Box Testing of Mass Assignment Vulnerabilities in RESTful APIs". ICSE 2023*

Automated Code-level Security Testing (2/6)

Let's start with approaches working at the system level (through API).


Metamorphic Testing for Web System Security

Nazanin Bayati Chaleshtari, Fabrizio Pastore, Member, IEEE, Arda Goknil, and Lionel C. Briand, Fellow, IEEE

Chaleshtari et al.[3]: Targets web applications endpoints, creating test cases using 100+ pre-defined templates implementing *metamorphic relationships (MRs)*, which are converted into executable Java code (using crawled URLs as entry points).

Honorable mention: EvoMaster[2] generates random test cases for web apps through their APIs (REST, GraphQL, and RPC) using genetic algorithms. Not security-specific.

Observation: They look like "special" vulnerability scanners. The difference is that they can return reusable test cases rather than just a report.

[3] *N. B. Chaleshtari et al. "Metamorphic Testing for Web System Security". In: IEEE Transactions on Software Engineering (2023)*
[4] *https://github.com/WebFuzzing/EvoMaster*

Automated Code-level Security Testing (3/6)

At the unit level, there is not much around. There is a small trend in third-party vulnerability testing, i.e., generating tests for an application that "imports" known vulnerabilities through dependencies (libraries, frameworks).

- **Goal**: check if a vulnerability is exploitable from the client code rather than directly.

Let's see the most relevant approaches.


Toward Automated Exploit Generation for Known Vulnerabilities in Open-Source Libraries

SIEGE[5]: Targets Java projects. It evolves a population of JUnit test cases starting from any client class (with a genetic algorithm) with the goal of producing one that reaches a specific vulnerable method in a dependency (exact location given).

- Fitness function: based on the project full call graph (including dependencies) and execution traces of the candidate test cases.
- **Drawback**: no assertions, only reachability tests!

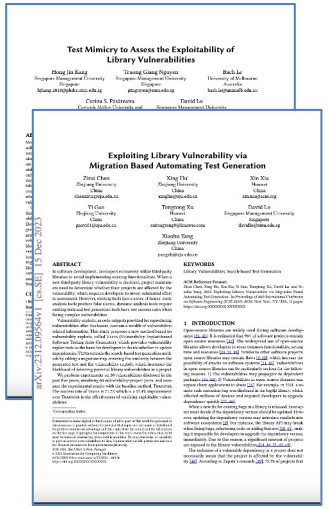
[5] *E. Iannone, D. D. Nucci, A. Sabetta, and A. De Lucia. "Toward Automated Exploit Generation for Known Vulnerabilities in Open-Source Libraries". In: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC). 2021.*

Automated Code-level Security Testing (4/6)

At the unit level, there is not much around. There is a small trend in third-party vulnerability testing, i.e., generating tests for an application that "imports" known vulnerabilities through dependencies (libraries, frameworks).

- **Goal**: check if a vulnerability is exploitable from the client code rather than directly.

Let's see the most relevant approaches.


Test Mimicry to Assess the Exploitability of Library Vulnerabilities

Exploiting Library Vulnerability via Migration Based Automating Test Generation

Transfer[6]: **Given a known security test case** belonging to the vulnerable library, it creates an "equivalent" test (i.e., replicates the same program state) that starts from the client code instead. Uses a genetic algorithm.

Vesta[7]: Similar to Transfer but instead of replicating program state of the starting test case, it builds new tests using **known exploits of the library vulnerability** (mined online) as test input data. Uses a genetic algorithm.

Both outperform SIEGE in the same set of known vulnerabilities.

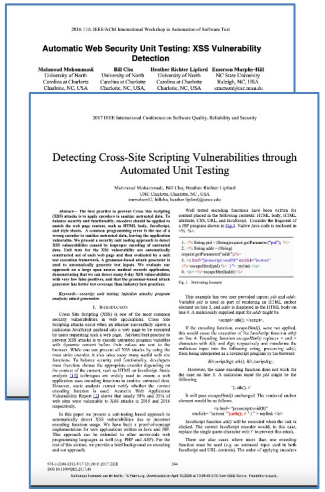
[6] *H. J. Kang et al. "Test mimicry to assess the exploitability of library vulnerabilities". ISSTA 2022.*
[7] *Z. Chen et al. "Exploiting Library Vulnerability via Migration Based Automating Test Generation". ICSE 2024.*

Automated Code-level Security Testing (5/6)

Limitation: Dependency testing cannot find new vulnerabilities!

- It can only confirm new ways to exploit a **known vulnerability**.
- It is more meant for risk assessment and dependency management.
- Very relevant, but it might not be what developers want.

We are looking for something working for "firsthand" code.


Automatic Web Security Unit Testing: XSS Vulnerability Detection

Detecting Cross-Site Scripting Vulnerabilities through Automated Unit Testing

Mohammadi et al.[8,9]: Creates unit test cases for **Java Server Pages to discover XSS vulnerabilities**. It runs a *taint analysis* on the JSP page and generates test inputs using pre-defined XSS attack grammars. A test confirms an XSS if the resulting HTML page displays a specific title.

- **Drawback**: Only for JSPs (not much unused today for web apps).

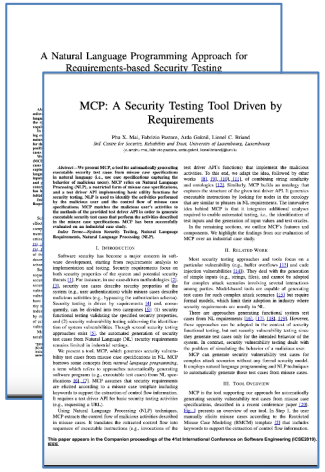
[8] *M. Mohammadi et al. "Automatic Web Security Unit Testing: XSS Vulnerability Detection". AST 2016.*
[9] *M. Mohammadi et al. "Detecting Cross-Site Scripting Vulnerabilities through Automated Unit Testing". QSR 2017.*

Automated Code-level Security Testing (6/6)

Limitation: Dependency testing cannot find new vulnerabilities!

- It can only confirm new ways to exploit a **known vulnerability**.
- It is more meant for risk assessment and dependency management.
- Very relevant, but it might not be what developers want.

We are looking for something working for "firsthand" code.


A Natural Language Programming Approach for Requirements-based Security Testing

MCP: A Security Testing Tool Driven by Requirements

Mai et al.[10,11]: Translates **misuse cases** into security test cases by leveraging a pre-defined **ontology** and a string similarity search to select the methods to invoke. The test assumes the vulnerability is discovered if all the steps of the misuse case are reproduced.

- **Drawback**: Need well-defined misure cases (very unlikely).

Moral of the story: We need more automated generation techniques!

[10] P. X. Mai et al. "A Natural Language Programming Approach for Requirements-Based Security Testing". ISSRE 2018.
[11] P. X. Mai et al. "MCP: a security testing tool driven by requirements". ICSE 2019.

Automated Exploit Generation

Security Tests and Exploits

Another popular term in this domain is **Vulnerability Exploitation**.

What's an exploit? Is it like a security test case?

Many definitions around (Cisco, TrendMicro, Rapid7, etc.). A convenience definition: "*any piece of software that takes advantage of one or more vulnerabilities in an asset (e.g., an application) to cause harm.*"

- Harm: Spawn a root shell, execute code remotely, steal credentials.

Exploit is a fluid concept: it ranges from malicious HTTP requests to *armed binaries (*equipped with a *payload)* that might spawn a root shell.

The boundary with security tests is blurry. One possible distinction:

- Exploit goal: demonstrating that a vulnerability can **be exploited to cause harm**!
- Security test case goal: showing that the tested code **has a vulnerability**.

As far as what concerns us in this talk, **it does not really matter**.

Automated Exploit Generation

Often, **Automated Exploit Generation** (AEG) build on top of other security testing approaches: static analysis, vulnerability scanners, fuzzing, etc.

Some noteworthy techniques are (not exhaustive):

- AEG[1] for memory-related bugs in C/C++ code.
- FlowStitch[2] for memory-related bugs in C/C++ code.
- FUGIO[3] for Object Injection in PHP code.
- Chainsaw[4] for SQLi and XSS in PHP code.
- EVIL[5] for generating shellcodes (payload) for Linux IA-32 (Intel x86 32 bits).
- ExploitGen[6] for generating shellcodes (payload) for Linux IA-32 (Intel x86 32 bits).

[1] *T. Avgerinos et al. "AEG: Automatic Exploit Generation". NDSS 2011.*
[2] *H. Hu et al. "Automatic generation of data-oriented exploits". SEC 2015.*
[3] *S. Park et al. "FUGIO: Automatic Exploit Generation for PHP Object Injection Vulnerabilities". USENIX Security 2022.*
[4] *A. Alhuzali et al. "Chainsaw: Chained Automated Workflow-based Exploit Generation". CCS 2016.*
[5] *P. Liguori et al. "EVIL: Exploiting Software via Natural Language". ISSRE 2021.*
[6] *G. Yang et al. "ExploitGen: Template-augmented exploit code generation based on CodeBERT". JSS 2023.*

Catalogs of Exploits

The de-facto standard of exploits is undoubtedly ExploitDB. The database contains both proofs-of-concept and shellcodes.

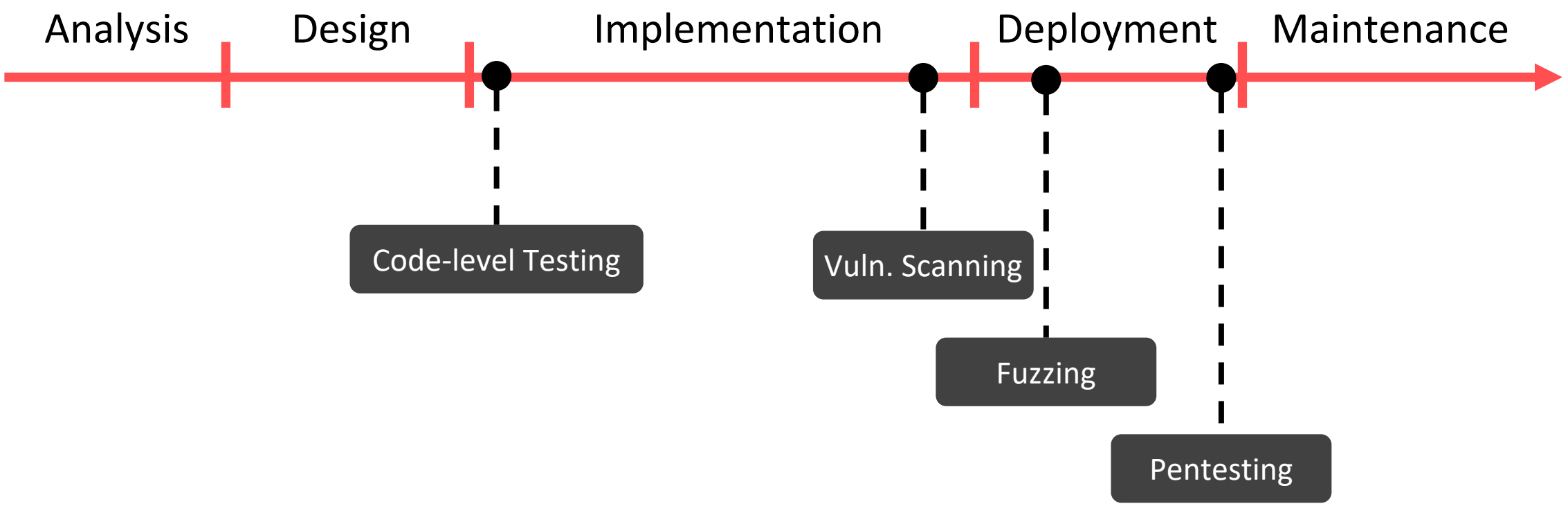
- https://www.exploit-db.com/
- https://gitlab.com/exploit-database/exploitdb

Other catalogs (different levels of maturity):

- Metasploit (Rapid7 Vulnerability & Exploit Database): https://www.rapid7.com/db/?type=metasploit
- AttackerKB (by Rapid7): https://attackerkb.com/
- Symantec Attack Signatures: https://www.broadcom.com/support/security-center/attacksignatures
- Shell-Storm: https://shell-storm.org/shellcode/index.html
- CISA's Known Exploited Vulnerabilities Catalog: https://www.cisa.gov/known-exploited-vulnerabilities-catalog
- SecLists (archive of security mailing lists) https://seclists.org/
- Microsoft Security Advisories (up to 2018): https://learn.microsoft.com/en-us/security-updates/securityadvisories/securityadvisories

Wrap-up

When Dynamic Testing Can Be Done



VERY simplified model.

Usages of Security Tests and Exploits

Security tests and exploits can help:

- Find new vulnerabilities.
- Demonstrate the exploitability of vulnerabilities (proofs-of-concept).
- Assess their severity (exploits more than “simple” tests).

More interesting usages

- Localize **commits that contributed to the introduction of vulnerabilities**. Security tests can triangulate the introduction moment.
  - Currently, they’re based on static mining software repository methods.
- Validate the security patches generated with AI-based repair approaches (*Automatic Vulnerability Repair*).
  - Rather popular research trend nowadays.

Limitations and Open Challenges

Many open challenges

- A few solutions for 3rd code testing and almost no solutions for "firsthand" code.
- No solutions for assessing the validity of generated tests/exploits.
- No empirical studies on the use of code-level security testing (how much, why, …).
- No attention to regression testing and maintenance of tests/exploits.
- Lack of datasets to validate the proposed techniques.

Research on automated security testing is in its infancy.

Possible reason: It is often believed that vulnerability scanners and fuzzers are **enough**.

Security Testing… for the Cloud

From the developer's perspective, API security in MSA applications is key.

- Top 10 API risks: https://owasp.org/API-Security/editions/2023/en/0x11-t10/
- Most issues are related to access control, as confirmed in the "issue taxonomy" by Waseem et al. (followed by issues with encryption and secure communication).[1]

A survey of 106 developers confirms that much attention is given to handling access tokens, but no **security testing techniques are known**.[2]

- Only a few exist (very recent): Micro-fuzz[3], Wang et al.[4], both based on fuzzing.

Direction: More cloud-specific research is needed!
API testing is the road to take for the time being.

[1] *M. Waseem et al. "On the Nature of Issues in Five Open Source Microservices Systems: An Empirical Study". EASE 2021.*
[2] *M. Waseem et al. "Design, monitoring, and testing of microservices systems: The practitioners' perspective". JSS 2022.*
[3] *P. Di et al. "MicroFuzz: An Efficient Fuzzing Framework for Microservices". ICSE-SEIP 2024*
[4] *W. Wang et al. "Zero-Config Fuzzing for Microservices". ASE 2023.*

Further Readings

Survey studies

- A survey on the main families of security testing techniques by Felderer et al.: LINK
- Two surveys comparing the key characteristics of several existing security testing approaches by Shahriar and Zulkernine: LINK, LINK
- Systematic Literature Review on search-based security testing with meta-heuristics by Ahsan and Anwer: LINK

Empirical studies

- Comparison of the efficacy of different automated testing approaches (3 SAST, 2 DAST, 2 manual testing) with a team of five researchers and 63 graduate-level students by Elder et al.: LINK
- Analysis of 481 instances of JUnit tests for Spring Security-based authentication functionalities from 53 open-source Java projects by Gonzales et al.: LINK

Third-party vulnerability testing

- An approach similar to Transfer but with ChatGPT by Zhang et al.: LINK
- Vuleut, another ChatGPT-based approach by Gao et al.: LINK

Questions?