Università degli Studi di Salerno

Dipartimento di Informatica

Tesi di Laurea Magistrale in
Informatica

Catch Me if You Can: Toward Automatic Exploit Generation of Known API Vulnerabilities

Relatori
Prof. Fabio Palomba
Prof. Andrea De Lucia

Candidato
Emanuele Iannone
Matr. 0522500588

Anno Accademico 2019–2020

Abstract

Modern software applications make an extensive use of Open-Source Software (OSS) components, forming the 35–75% of the total codebase. This scenario opens to serious security threats, mainly because developers rely on non-updated versions of the libraries affected by software vulnerabilities; for this reason, numerous detection tools have been developed though years to assist developers in finding and mitigating these known issues. While they provide a great support, they are still immature in assessing the exploitability of the detected vulnerabilities, *i.e.,* establishing whether they are feasible in practice. In this work we propose a novel automatic exploit generation approach based on genetic algorithms that tries to "catch" a given vulnerability with an artificially created exploit. This technique has been implemented as an extension for EvoSuite – a well-known automatic JUnit test generation tool – and applied a restricted set of toy projects that relies on vulnerable versions of widely used OSS Java libraries and frameworks. The preliminary findings show promising results that, however, need to be further assessed in the context of larger scale empirical studies.
Index Terms — Genetic Algorithms, Exploit Generation, Vulnerabilities.

Contents

Chapter 1

Introduction

1.1 Context and Motivation

Code reuse is one of the key practises in modern software development [1] that allow developers to reduce the implementation costs and time-to-market. The preferred form is definitely the *black-box reuse* by the means of *third-party libraries*, *i.e.,* a set of distributable modules and functionalities to be included within other software projects. This is supported by the fact that they are (i) pretty well tested and documented and (ii) often distributed along with the source code (*i.e.,* open-source software, or simply OSS), meaning that their *client* developers (namely, the developers who include them into their projects), if desired, could make their own contribution.

Nowadays, most large companies and open-source foundations, like GOOGLE, APPLE or APACHE, are providing hundreds of APIs to foster the development of software for expanding their ecosystems [2]; as a consequence, some recent estimates say that 80% to 90% of software products on the market include some form of OSS components, while other estimates say that these components' distribution depends on the kind of application: for commercial ones we have a distribution of 35% of total codebase, while for internal use application we reach the 75% [3]. Unfortunately, as any software system, these third-party libraries may suffer from defects, and the implied negative effects are more harmful than common systems for two great reasons: (i) the potential propagation of a defect to other indirect client projects — in the so-called **ripple effect** — and (ii) the impossibility to **immediately and directly** fix the issue but waiting for the next release or hotfix by the authors (or other contributors for OSS). One of the main form of defect that regularly affect third-party libraries are vulnerabilities [4, 5], that we can informally describe as software portions exposed to potential attacks against the security policy of the system, typically defined via the CIA triad, *i.e.,* confidentiality, integrity, and availability. Vulnerabilities, together with the two aforementioned negative effects, are particularly tricky, especially because they could be easily propagated to all direct and indirect clients, as happened for the infamous HEARTBLEED[1] bug where a naive vulnerability in OPENSSL (a well-known OSS library that implements the TLS protocol) impacted on almost half-million certified websites (17% of the total) or with the incident at EQUIFAX where an known vulnerability in APACHE STRUTS framework affected their U.S. website and exposed the entire back-end system to a serious data breach in 2017 that led to billions of dollars of total damages[2].

As time goes by, more and more vulnerabilities of very popular OSS libraries are being discovered and publicly disclosed in *Common Vulnerabilities and Exposures* (CVE) databases, such as the *National Vulnerability Database*[3] (NVD). This growing trend made in 2013 to label *Using components with known vulnerabilities* as one of the *OWASP Top 10 Web Application Security Risks*[4]. To tackle this problem, numerous weakness and vulnerability detection tools have been made available, either as open-source or as commercial products [6, 7, 8, 9, 10]; basically, they analyse a given project in search of weak code patterns — *e.g.,* using the `Statement` class instead of `PreparedStatement` class when using the JDBC API, exposing to possible SQL *Injection* attacks — or usages of known vulnerable OSS components — *e.g.,* adding a vulnerable version of TOMCAT into the MAVEN project's *pom.xml* file. The detected weaknesses could be mitigated by applying relatively easy refactoring actions that remove the bad code pattern(s) — *e.g.,* replacing the `gets()` function call with the more secure `fgets()` to avoid possible buffer overflow-based attacks. Unfortunately, as anticipated, the detected OSS vulnerabilities cannot be directly fixed by the client developers, so the easiest and most commonly applied mitigation consists in the update of the vulnerable library into an available non-vulnerable version. This solution is acceptable for software that is still under development and far from its first release because all

[1]`https://heartbleed.com/`
[2]`https://tinyurl.com/v9x2dq5`
[3]`https://nvd.nist.gov/`
[4]`https://owasp.org/www-project-top-ten/`

the necessary adaptations in the application code can be performed as part of the normal development activities; on the other hand, updating a library for software that is already in production or near its first release is a risky action because it may force changing several API calls (*e.g.,* when updating to a new major release) and running the regression tests, so requiring extra time and effort. This scenario is better explained in [11] where developers perceive the library updates as extra workload and responsibility, making them very reluctant.

Hence, a tool that simply checks if the included dependencies are of a vulnerable version would not concretely support the developers in the mitigation step, because they would be required to manually evaluate the concrete risk of a large amount of libraries. In other words, the inclusion of a vulnerable library is a necessary but not sufficient condition for the actual exposure. Therefore, a good detection tool should provide fine-grained analyses in order to reduce the number of false positives (*i.e.,* non-vulnerable dependencies incorrectly flagged as vulnerable) for the purpose of reducing the entire mitigation workload.

The main fine-grained analysis is the *static reachability* one, that checks whether from a given project one or more known vulnerable OSS constructs (*i.e.,* any programming construct, such as a statement, a branch, a method, a class, etc., belonging to an OSS component) can be statically reached in the global Control Flow Graph (CFG); while this approach is more robust than the simple inclusion analysis, it does not guarantee that a reachable vulnerable construct can be exploited to carry out a concrete attack, for instance, the construct might be guarded by a good sanitization procedure which prevents it to receive any kind of malevolent input.

In order to overcome this limitation, certain tools apply a *dynamic reachability* analysis, alongside the static one, by relying on the available test suite and/or on any kind of runtime information; this solution, among other things, takes into account dynamically loaded code, such as the one called though JAVA REFLECTION [12] or the one executed by frameworks or containers (*e.g.,* SPRING or TOMCAT), *i.e.,* the code that gives the control to our application, according to the Inversion of Control (IoC) principle. In addition, the dynamically loaded code may call an extra amount of code constructs not caught in the initial static analysis (such as in the case of framework code). For this reason, tools like ECLIPSE STEADY [10] further expands these two analyses by applying an additional static reachability on top of the dynamic analysis results.

The main limitations of all the forms of dynamic analyses is the uncertainty of having good test suites and/or runtime information to use for driving the process, making the static reachability the only possible solution. Moreover, to the best of our knowledge, the currently available tools are not capable of assessing the *degree of exploitability* of the vulnerable constructs, making impossible to determine whether a certain construct is exploitable in practise.

Recently, there has been some initial efforts in the field of Search-Based Software Engineering regarding the assessment of vulnerabilities. Jan et al. [13] proposed an evolutionary algorithm that tries to generate a set of malicious user inputs (represented by *form input data*) that will be transformed into a set of potentially malicious XML messages that will carry out XML *Injection* attacks in web applications. Despite its conceptual simplicity, this solution has some limitations:

1. it only works on a specific type of vulnerabilities (namely, XML *Injection* weaknesses[5]);
2. it focuses on a defined set of malicious XML messages only, to be prepared a priori;
3. it treats the whole system under test (SUT) as a black-box, making it unaware of the concept of vulnerable construct due to its lack of knowledge about the source code or bytecode.

One last, but not least, issue is the lack of developers' awareness of the presence and the risk of the vulnerable dependencies in their code [11].

All these things together raise the need of a better support to the risk assessment part of the entire vulnerability analysis cycle.

1.2 Proposal

This work provides a new search-based assessment technique called **SIEGE** (Search-based automatIc Exploit GenEration) that tries to generate exploits against a software project that includes a set of known OSS vulnerabilities. In this way, if the algorithm succeed generating an exploit for a target vulnerability, the latter will be considered more exposed to dangers that others, requiring more attentions for the subsequent mitigation actions; on the other hand, if the algorithm fails in the generation, we could say that the target might be less exposed to risks.

[5]`https://cwe.mitre.org/data/definitions/91.html`

This approach is implemented as an extension of EvoSuite [14], a well-known automatic JUnit test generation tool that offers a quite rich infrastructure for the development of evolutionary algorithms. The extension consists of adding an ad-hoc coverage criterion with some other tweaks to make EvoSuite generating exploits instead of classical test cases.

Then, the approach is empirically validated on a set of artificially crafted projects that include really existing OSS vulnerabilities selected from the dataset of Ponta et al. [15].

The preliminary findings show promising results that corroborate the feasibility of the proposed methodology.

1.3 Structure of the Thesis

The thesis is structured as follows. Chapter 2 provides some background information on vulnerabilities, search algorithms and the EvoSuite tool; Chapter 3 shows the novel technique in detail; Chapter 4 shows the empirical validation and its main findings; Chapter 5 concludes the thesis by presenting the lessons learnt and the future directions.

Chapter 2

Background and Related Works

This chapter starts describing how third-party libraries evolve through time. Then, it proceeds on software vulnerabilities, by giving some basic definitions, showing their lifecycle and explaining their presence in OSS libraries. Finally, it concludes presenting the state-of-the-art on automatic test case generation and describing a notable tool: EvoSuite.

2.1 Third-party Libraries Evolution

Third-party libraries inevitably evolve through time [16]. Ideally, their public interface — that forms their API (Application Programming Interface) — should keep the backward compatibility across versions, so that each client application (*i.e.,* the software that includes them) would not be impacted by their internal changes. However, as shown by Dig and Johnson [17], this does not happen in practise since third-party libraries often must change their API as well, requiring client developers to spend additional effort and risking to introduce bugs.

Some API changes affect the syntactic structure of the client application, meaning that the compiler will fail its job, *e.g.,* when a public library method changes its name or its argument list; other changes impact on the semantics of the applications, meaning that, despite the build succeed, the client will functionally behave differently at runtime. In both of the cases we are referring to *breaking API changes*, that cannot preserve the backward compatibility. On the other hand, there are certain changes, called *non-breaking API changes*, that have absolutely no impact on clients, meaning that the backward compatibility is preserved, *e.g.,* when a private library method is refactored to remove a code smell.

Although preserving the API is the best choice, library developers are often forced to break it for adding new features, fixing bugs or improving the general internal quality. To mitigate the related risks, library developers try to adopt various best practices for keeping their APIs the most stable as possible, such as by using very long *deprecate-replace-remove* cycle — a technique for gracefully degrading old APIs through (i) the coexistence of both old and new API and (ii) the encouragement of clients on switching to the new one before the old one is removed in future releases.

Dig and Johnson showed that over 80% of breaking API changes in third-party libraries are caused by refactoring and that he top-3 of most breaking refactorings are *Move Method*, *Move Field* and *Delete Method*. This result should not be a surprise since the Martin Fowler's definition of refactoring [18] concerns the external behaviour preservation only, not the public interface of refactored components; thus, API changes, as long as they keep the external behaviour intact, are tolerated by refactorings.

All of this does not foster clients in regularly upgrading the libraries they include into their projects. This fact is more evident in the work of Kula et al.[11] where over 80% of projects hosted on GitHub keep their dependencies outdated.

2.2 Software Vulnerabilities

A software vulnerability, intuitively, is a logic or configuration flaw that exposes a software system to attacks, such as data breaches, data manipulation, denial of a service, and so on. There is not a precise definition of what is software vulnerability and what is not, and basically this is left at discretion of a product owner [6]; however, it is a diffused practise to define it on the concept of weakness.

Definition 2.2.1. A (software) **weakness** is a type of defect within a software product that, in proper conditions, could contribute to the introduction of security threats within a product[7].

It appears that a weakness per se does not necessarily expose the system to threats, but some precautions should be taken to avoid potential damages.

[6] `https://cve.mitre.org/about/terminology.html`

[7] `https://cwe.mitre.org/documents/glossary/index.html#Weakness`

Definition 2.2.2. A (software) **vulnerability** is an occurrence of a weakness (or multiple weaknesses) within a software product, that can be used (*i.e.,* exploited) by a party (*i.e.,* the attacker) to cause the product to modify or access unintended data, interrupt proper execution, or perform incorrect actions that were not specifically granted to the party[8].

Thus, a vulnerability is a concrete threat for the system, making it exposed to malicious usages known as *exploits*.

Definition 2.2.3. An **exploit** is a procedure that uses a vulnerability to negatively impact on confidentiality, integrity, or availability of the target product[9].

Definition 2.2.4. An **attack vector** is the mean/way by which an attacker can exploit a vulnerability.

Definition 2.2.5. The **attack surface** is the set of all possible attack vectors against a product.

Let's consider a Tomcat-based e-commerce web application that implements the Data Access Object (DAO) pattern for accessing the back-end database. One of the DAO classes uses the `Statement` class to define a query that fetches a product from the catalogue by using the user-supplied identifier. This query is executed when doing a `GET` request at: `/product?id=20`. The results of the query is directly put into the `HTTP` response, without any other manipulation. If the user agent (*e.g.,* the browser) performs the request `GET /product?id=1'or'1'='1`, the server will improperly read all the products in the database, invalidating the rest of the possible `WHERE` clause. This happened because the application adds the user-supplied data directly into the query without properly removing certain reserved characters or keywords (such as, `#`, `'`, `UNION`, etc.), allowing an attacker to have full control of the query, and so the entire database. Thus:

– The 'weakness' is caused by the improper neutralization of the special SQL elements;
– The 'vulnerability' affects this specific web application and it is caused by the said weakness;
– the 'exploit' consists in the possibility for a normal user to read data from the database without the required rights;

[8]`https://cwe.mitre.org/documents/glossary/index.html#Vulnerability`
[9]`https://nvd.nist.gov/vuln`

– the ‘attack’ surface is a set of all malicious `GET` requests.

This vulnerability can be patched by fixing the underlying weakness, that in turn can be removed with various alternatives, for example:

– replacing `Statement` class with `PreparedStatement` class that automatically escapes special SQL elements;
– implementing a validation procedure (preferably server-side) that escapes the special SQL elements.

Now suppose that the web application uses a *root* connection to access the database server independently from the effective user. In this scenario, any user will have the access rights for every operations on every schemes present there. This additional weakness creates a new and more dangerous vulnerability that allows an attacker to fully explore the database content or, even worse, delete it all.

These two described weaknesses are very common in practise, and they are further detailed in two CWE entries:

1. *CWE-89: Improper Neutralization of Special Elements used in an SQL Command (’SQL Injection’).*
2. *CWE-250: Execution with Unnecessary Privileges.*

CWE (Common Weaknesses Enumeration) is a uniform catalogue of both software and hardware weakness types. Each CWE entry has a unique identifier and provides some demonstrative examples of a weakness, along with the main implications and some common mitigation actions. Similarly, CVE (Common Vulnerabilities and Exposures) is a uniform catalogue of known vulnerabilities. Each CVE entry has a unique identifier and refers to a specific vulnerability that affects a specific software or hardware system. For example, the CVE-2014-0160 refers to the *Heartbleed* bug, caused by an erroneous implementation of TLS protocol in OpenSSL introduced in 2012 and publicly disclosed (*i.e.,* when the product owner recognise the threat that affects its system) in 2014, and based on the weakness *CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer*, Each CVE entry is also listed into the National Vulnerability Database (NVD), that extends them with additional details, such as fix information, severity scores, and impact ratings.

2.2.1 Lifecycle

How do developers introduce vulnerabilities in their code? Are they really aware of their presence? Currently, there is poor knowledge on how vulnerabilities actually manifest themselves in a software project, how long they tend to live or how they are removed. Iannone et al. [19] are currently investigating the lifecycle of software vulnerabilities from a dataset of 1195 vulnerabilities of 62 types (weaknesses) across 616 GitHub projects, namely:

– *When* they are introduced;
– *Under which* circumstances they are introduced;
– *How long* they survive;
– *How* they get removed.

The study selected the vulnerabilities from NVD for which both the *fixing commit* (*i.e.,* the commit that officially patches the vulnerability) and the GitHub repository are available. Typically, a fixing commit could be identified by the looking for the presence of certain keywords in the commit message, such as 'fix', 'patch', 'CVE', etc., or with a link to an open issue related to a security defect.

Starting from a fixing commit, they applied the SZZ [20] algorithm to 'go back' to the *vulnerability-inducing commits.* Basically, SZZ applies the `git blame` command to each of the modified files in the fixing commit to obtain the commits which last modified these files. The idea behind SZZ is that the obtained commits are responsible for the seeding of the vulnerability. The entire lifespan from the first to last vulnerability-inducing commit is called *introduction period*; the last vulnerability-inducing commit marks the turning point of the official introduction of the vulnerability in the code, starting the *survival period*, that lasts until the fixing commit. Moreover, from the metadata of a vulnerability-inducing commit, it is possible to get additional information about the context in which the vulnerability was introduced, such as the developer, the timestamp, etc.

The study shows that:

– Introducing a vulnerability does not require a high number of commits (a median of 2);
– vulnerability-inducing commits are size-limited (an average of 3.63 touched files). This goes against the common knowledge on defect-prone commits, that typically touches a high number of files;

- vulnerability-inducing commits tend to create files that are not vulnerable yet, and they become flawed in few commits (a median of 3);
- vulnerability-inducing commits may have different goals ("new feature", "bug fixing", "enhancement" or "refactoring"). Despite the most common goal being "new feature", a non-negligible amount of commits are for "refactoring";
- most vulnerabilities are introduced near a deadline (even the day before) and far from the project startup;
- expert developers with high workloads are more prone to introduce vulnerabilities;
- vulnerabilities tend to be very persistent (in median, for about 1000 commits and across 5 years);
- some vulnerabilities are both introduced and removed by the same developer;
- the patches range from simple program transformation to complex restructuring.

The reason why vulnerabilities tend to live for long periods of time is still unknown, but it may be caused by the developers' lack of awareness about their presence; this reason highlights the urgency and need of just-in-time detection and assessment techniques, that would support their decisions with respect of weaknesses and vulnerabilities.

2.2.2 Vulnerabilities in OSS Libraries

As previously said in chapter 1, *Using components with known vulnerabilities* has been added into *OWASP Top 10 Web Application Security Risks* since 2013. The Open Web Application Security Project (OWASP) is a foundation whose goal is to improve the security of software[10]; among the others, its main project is the *OWASP Top 10* — a globally recognized list of the most common and dangerous design and implementation choices that affect the security of web applications. Currently, the Top 10 places *Using components with known vulnerabilities* at the ninth position, while on the podium we can see:

1. *Injection* (of any kind) that occurs when untrusted data is sent to an interpreter as part of a command or query, leading it to execute unintended operations without proper authorization.

[10]`https://owasp.org/`

2. *Broken Authentication* refers to poorly implemented authentication and session management functions.
3. *Sensitive Data Exposure* refers to the lack of protection — such as with encryption or hashing — of sensitive data.

Theoretically speaking, resolving the problem of vulnerable OSS components is simple: one could just update all project's dependencies as soon as new versions come out. Unfortunately, this does not happen in practise, mainly because any library update is considered a risky action and developers are reluctant in applying them [11], especially when near a deadline. Things get even more complicated when considering the fact that disclosed vulnerabilities are often expressed as short and high-level textual descriptions, requiring:

1. a deep knowledge of the application;
2. a manual assessment activity for finding any potential exposures.

Moreover, both false positives (*i.e.,* non-vulnerable dependencies incorrectly assessed as exploitable) and false negatives (*i.e.,* vulnerable dependencies incorrectly assessed as non-exploitable) have bad consequences: the formers determine a waste of effort, especially for dealing with the ripple effect of the updates, while the latters expand both the attack surface and the time window in which the application remains exposed to attackers. Therefore, the manual assessment mode is often avoided.

Let us make order of all the above concepts. In the context of OSS vulnerabilities, the vulnerabilities are addressed with three main actions:

– *Detection*, whose goal is to analyze a project's dependencies in search of the vulnerable ones.
– *Assessment*, whose goal is to establish the risk of all the detected vulnerable dependencies.
– *Mitigation*, whose goal is to provide support for applying the correct patch.

The main effort are on the provision of automatic approaches to allow developers to take informed decisions and reduce their workload. The following sections will show their state-of-the-art and current challenges.

Detection

There are numerous OSS vulnerabilities detection tools, such as OWASP Dependency Check [6] and Whitesource [7]. Basically, they check if any known vulnerable OSS library is included within projects dependency graph — built from any package manager manifest file, such as *pom.xml*, *package.json*, *Gemfile*, etc. — and in that case a security alert is sent. We call this simple solution *inclusion analysis*. This approach provides good performance, is conceptually simple and can be easily integrated into other tools or platforms, as was done for Whitesource within GitHub[11]; however, it fails in terms of precision (*i.e.,* it leads to many false positive), asking the developers to manually inspect the concrete risk — that may not be present at all — of each security alert.

To overcome these limitations, Eclipse Steady [10] (formerly, *Vulas*) applies fine-grained analyses, based on the *reachability* of the program constructs (such as methods), in order to reduce the number of false positives. The core of Steady lies on the assumption that a vulnerability can be uniquely described by considering the set of program constructs that were modified, added, or deleted in its fixing commit.

Steady builds a *Bill of Material* of vulnerable constructs starting from a dataset of known OSS vulnerabilities with their linked repositories and fixing commits — such as the one from Ponta et al. [15]. Once this data structure is built, Steady is ready to run its reachability analyses:

– `a2c`, a static reachability analysis that tries to reach any vulnerable construct starting from the application code.
– `upload`, a dynamic reachability analysis that tries to reach any vulnerable construct from the execution traces of all JUnit tests (if any).
– `t2c`, a static reachability on top of the results of `upload` that tries to reach any vulnerable construct from the dynamically reached constructs.

Figure 2.1 graphically summarises the three main analyses of Steady.

One limitation of this approach occurs when a fixing commit does not change any code construct, but only lines in configuration files. In this way, the reachability analyses cannot be applied (because of the lack of vulnerable constructs), even though the simple inclusion analysis is still feasible. Moreover, vulnerabilities

[11] `https://docs.github.com/en/github/managing-security-vulnerabilities/about-alerts-for-vulnerable-dependencies`

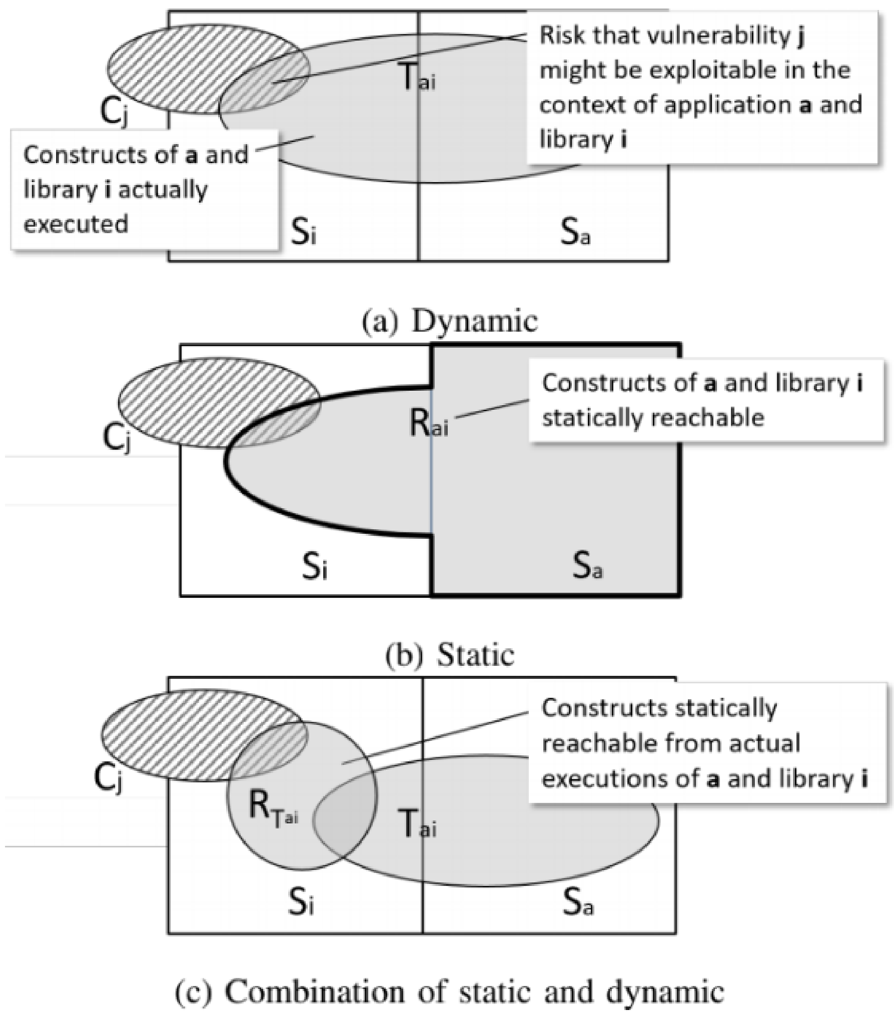

Risk that vulnerability j might be exploitable in the context of application a and library i
T_{ai}
C_j
Constructs of a and library i actually executed
S_i
S_a
(a) Dynamic
Constructs of a and library i statically reachable
R_{ai}
C_j
S_i
S_a
(b) Static
Constructs statically reachable from actual executions of a and library i
C_j
$R_{T_{ai}}$
T_{ai}
S_i
S_a
(c) Combination of static and dynamic

Figure 2.1: Steady detection

based on *CWE-502 Deserialization of Untrusted Data*[12], could be exploited with just the mere presence of certain weak classes in the classpath [13], regardless of both static and dynamic reachability of the affected constructs.

The Steady detection approach seems to be the right way to increase the awareness on vulnerabilities, however, the usage of yet-another analysis tool does not facilitate the developers, so more integrated solutions are required.

[12]`https://cwe.mitre.org/data/definitions/502.html`
[13]`http://frohoff.github.io/appseccali-marshalling-pickles/`

Assessment

Steady shifted the problem of exploitability to the reachability analysis, meaning that if the vulnerable code is reachable then it there is a way to exploit it. While this may be true for most of the cases, this does not always happen in practise.

Let us consider a vulnerable function `foo()` within the library `Foo`. Suppose that this function is only called by the application though the function `baz()`, that is aware of the security issue in `foo()` and avoids passing it any possible malicious input. In this scenario, the vulnerability would never be exploited because of the presence of `baz()`. A tool like Steady would still flag the library as vulnerable and suggest an unnecessary update (false positive), even thought the problem, from the point of view of the client function `baz()`, is solved.

Mitigation

The mitigation of OSS vulnerabilities should look for the most suitable non-vulnerable library version available, meaning that the update should both fix the problem and minimize the developers' workload, that translates into choosing the library that has the minimum impact on the entire application. Assuming the presence of at least two non-vulnerable versions to choose, we need some metrics that capture (i) how the application uses the library and (ii) the amount of changes from the current vulnerable version to any of the available ones;

Steady adopt a solution that takes into account these factors and uses the following metrics:

– *Callee Stability*, that measures how many directly called library constructs are preserved in the update.
– *Development Effort*, that measures how many calls are impacted by the update.
– *Reachable Body Stability*, an extension of Callee Stability for all the indirectly called library constructs.
– *Overall Body Stability*, a variant of Reachable Body Stability that considers non-called library constructs as well.

These metrics, similarly to the ones of Raemaekers, van Deursen, and Visser [21], are totally code- and usage-centric.

It would be good to consider additional metrics such as the *popularity* of the update, *i.e.,* preferring the most downloaded version, as done in [22, 23]. In any case, the metrics adopeted by STEADY act as mere predictors of the effort that a developer would invest on an update, without providing any practical support in the migration process — in fact, the risk of massively breaking the API by picking the wrong version is still alive. For this reason, we need library migration tools and techniques tailored on the context of software vulnerabilities [24].

2.3 Search-based Software Engineering

Search-based Software Engineering (SBSE) focuses on using search algorithms and techniques to solve Software Engineering-related problems [25]; in particular, Search-Based Software Testing (SBST) deals with testing-related problems, such as Test Case Minimization, Selection or Prioritization. Among these, the most prominent one is the Automatic Test Case Generation (ATCG) that aims providing a minimal-cost test suite that maximizes a set of test requirements, *e.g.,* covering the largest number of branches of a method with the smallest number of statements among all test cases.

The generation of test cases (sometimes, the test input data only) relies on search algorithms guided by an *objective function* that models a set of *test goals* (*a.k.a.,* test objectives) to derive the test suite that best fulfills the given goals. There are many kinds of search techniques, but in recent years the literature has been working on evolutionary approaches, in particular on *genetic algorithms* (*a.k.a.,* GAs).

2.3.1 Genetic Algorithms

A genetic algorithm is a meta-heuristic (*i.e.,* a high-level procedure that can be applied to a broad range of search problems) inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms.

Algorithm 1 summarises the general schema of a GA. It starts with an initial random *population* of *individuals* (*a.k.a.,* chromosomes), which are candidate solutions for the given problem and whose *genetic structure* depends on the specific problem representation. The population passes through different *generations* (iterations) that continuously *evolve* the individuals in order to get them close to the optimum value — with respect to the objective function, that in this context is also called *fitness function*. The evolution consists on applying different *genetic*

Algorithm 1 General schema of a Genetic Algorithm

```
    Input: FitnessFunction $F$, StoppingCondition $SC$
    Output: Individual $b$
 1: begin
 2:     $P \leftarrow$ Initial Random Population
 3:     repeat
 4:         Evaluate $P$ with respect to $F$
 5:         $P_1 \leftarrow$ Selection($P$)
 6:         $P_2 \leftarrow$ Crossover($P_1$)
 7:         $P_3 \leftarrow$ Mutation($P_2$)
 8:         $P \leftarrow P_3$
 9:     until $SC$ is met
10:     $b \leftarrow$ GetBestIndividual($P$)
11:     return $b$
12: end
```

operators on the individuals, namely:

1. *selection*, that chooses the best individuals — the higher the fitness score, the higher the probability to be chosen — for the creation of the next generation;
2. *crossover*, that crosses the genetic structure of a pair of *parents* to generate *offsprings*, that may be included in the next generation;
3. *mutation*, that randomly changes part of the genetic structure of the newly-formed offsprings, according to a certain probability.

The evolution may stop for a number of reasons, such as:

– the search budget (*e.g.,* running time) expires;
– the population reaches the convergence (*i.e.,* it has an optimal individual);
– the population stops making progresses, meaning that for some iterations the aggregate fitness score (related to the entire population) had no improvements.

Typically, a fitness function is expressed as a minimum function (*i.e.,* the optimum value is the function's absolute minimum). While some problems require the use of a single fitness function (*a.k.a.,* single-objective optimisation), others require multiple and often conflicting functions (*a.k.a.,* many-objective optimisation), demanding a redefinition of the concept of optimality to take into account multiple fitness scores — collected into a *fitness vector* — and their trade-offs. In many-objective optimisation problems we need the broader concepts of *Pareto dominance* and *Pareto optimality* [26].

Definition 2.3.1. An individual x *dominates* another individual y (also written $x \prec y$) if and only if the values of the fitness vector satisfy the following conditions: $\forall i \in \{1, \dots, k\}, f_i(x) \leq f_i(y)$ and $\exists j \in \{1, \dots, k\}, f_j(x) < f_j(y)$, where k is the number of fitness functions.

Definition 2.3.2. An individual x^* is *Pareto optimal* if and only if it is not dominated by any other individual in the space of all possible individuals (feasible region).

Definition 2.3.3. The *Pareto front* is the set of all Pareto optimal individuals.

In the context of ATCG, an individual (typically) models a test case that, as required by a common testing framework, is a method that exercises a portion of the production code and verifies a certain behaviour through the use of assertions. Thus, an individual can be described by two distinct parts:

1. the list of statements of the test code, that have to be compliant to a well-define pattern known as the *template*;
2. the actual values used in method calls, assignments and, possibly[14], assertions.

Let us consider the test method in Listing 2.1.

```
1 @Test
2 public void test01() {
3   Foo foo = new Foo(); {
4   Bar bar = new Bar(2); {
5   int r = foo.fantasticMethod(bar); {
6   assertEquals(20, r); {
7 }
```

Listing 2.1: A unit test implemented as JUnit 4 test method.

We can clearly distinguish the four parts of the template, that are:

1. `Foo foo = new Foo()` (Line 3) represents the part when the class under test (CUT) is instantiated;
2. `Bar bar = new Bar(2)` (Line 4) represents the part when the required objects are built before the method under test is called;

[14]As said, many ATCG techniques deals with the generation of test input data only, putting aside the oracle.

3. `int r = foo.fantasticMethod(bar)` (Line 5) represents the part when the method under test is called and its result is collected;
4. `assertEquals(20, r)` (Line 6) represents the part when the expected outcome is compared with the actual result of the method under test;

A possible representation as chromosome is the following:

```
$foo=Foo():$bar=Bar(int):$r=$foo.fantasticMethod($bar):assertEquals
    ↪(int,$r) @ 2, 20
```

Listing 2.2: The chromosome representation of the test method in Listing 2.1.

The first part before the '@' symbol contains the list of statements according to the described template, while the second part contains the actual values that completes the missing parts of the test code. The GA treats these two parts differently; for example, a mutation operation may change the constructor of `Foo` to another available one or change the number 20 to 30.

The test goals are often defined through *coverage criteria* (proper to white-box testing), such as branch, line or mutation. Thus, the fitness score(s) depends on how much an individual's execution trace is close to the aforementioned goals; for instance, if the goals are defined in terms of branch coverage, then the fitness score is computed on the number of control dependencies that separate the execution trace from the target branch (*approach level*) and on the variable values evaluated at the conditional expression where the execution diverges from the target (*branch distance*) [27].

2.3.2 Notable Implementations

The various forms of fitness functions are not the only variants that can be applied on GAs; indeed, there are numerous examples in literature of how malleable GAs are.

In many-objective formulation the number of (Pareto) optimal individuals increases exponentially with the number of objectives, making the search much more difficult (*i.e.,* the convergence is hardly reached) and acting similarly to a random one. Thus, domain-specific knowledge is required to impose some *preference criteria* among the set of non-dominated test cases. Panichella, Kifetew, and Tonella [28] proposed MOSA (Many-Objective Sorting Algorithm), which uses multiple fitness functions — one for each branch of the CUT — to generate the best set of Pareto optimal test cases that minimize all the fitness functions while preferring individuals that (i) cover, or are about to, yet uncovered targets and (ii) are smaller than its competitors. As consequence, the set of individuals candidate for reproduction is a subset of the entire Pareto front. In addition to these preference criteria, MOSA adds the *archive* technique, that consists of keeping a distinct non-evolving population containing the test cases that satisfy newly covered targets. After the final iteration, the test cases are picked from both the last population and the archive to form the final test suite.

The main limitation of MOSA is that it treats all coverage goals as independent objectives, when, actually, there exist structural dependencies among them that should be considered when deciding which ones to optimise; for example, a certain branch could be satisfied if all branches that hold a control dependency on it have already been covered (a necessary but not sufficient condition). To overcome this limitation, the same main author of MOSA proposed DynaMOSA (Dynamic MOSA) [27] that is able, in each iteration, to *dynamically select* the targets whose control dependency holders have been covered in earlier iterations and to ignore the other targets. This idea makes the search more effective and efficient in situations where there are a lot of dependencies among coverage goals, *e.g.,* the branches of a method.

A quite different optimisation approach was proposed by Arcuri with MIO (Many Independent Objectives) [29], an evolutionary algorithm that works well for hundreds/thousands of independent (though not incompatible) goals. For each goal it keeps one population — called *island* — that evolves independently of the others (namely, the individuals of an island are only evaluated against the respective fitness function). The reproduction mechanism is not based on genetic operators (this is why MIO is not classified as a GA), but rather on sampling individuals either randomly or from other islands (*i.e., migration*). Whenever a goal is covered, its island stops evolving and only the best individual survives; at the end of the entire search, the best individuals from each island form the final solution.

The MIO approach can be combined with GAs, creating the so-called class of *co-evolutionary* algorithms, that add to MIO the standard genetic operators (though with some variations) along with the periodical migrations. An example of a co-evolutionary algorithm is COMIX (Cooperative Co-evolutionary Algorithm for XMLi) of Jan et al. [13] that was used for the automatic generation of malicious user inputs that exploit XML *Injection* vulnerabilities in web applications. An important difference between COMIX and the other aforementioned algorithms is that the test goals are not based on the typical coverage measures from white-box testing but, instead, they model a predefined set of malicious XML messages, making the strategy totally black-box.

Fraser and Arcuri, in the context of EvoSuite [14], proposed the *whole suite* approach that changes the point of view: instead of evolving a population of test cases, it evolves a population of test suites, and the single fitness function models all the test goals at once. The fitness score of a single test suite is an aggregation of the fitness sub-scores of each composing test cases. Moreover, the EvoSuite's algorithm has a preference for small test suite, — namely, it tries to find the test suite with the minimum number of total statements among all individuals. EvoSuite is compatible with multiple coverage criteria, such as branch, method, line, def-use, mutation, exceptions, etc. and it can also *generate regression assertions* for each test case of the final test suite.

2.4 Automatic Test Case Generation with EvoSuite

EvoSuite [14] is an automatic JUnit test case generation tool written in Java for Java projects. It works on the bytecode level and extracts all the needed information via Java Reflection, meaning that it does not strictly require the source Java code of the SUT. This makes it valid for any bytecode-compliant language, *e.g.,* Scala or Kotlin. EvoSuite is implemented with a client/server architecture, where the server manages all the execution and collects the statistics, and the client does the actual generation and tests execution. EvoSuite is available as a command-line tool and as a plugin for Eclipse, IntelliJ IDEA and Maven.

Before starting any generation, EvoSuite instruments the bytecode of the SUT with additional statements for two main purposes:

1. collecting all necessary information (*e.g.,* execution traces, branch distances) to calculate fitness values;

Algorithm 2 General workflow of EvoSuite

```
    Input: CrossoverProbability $XP$, StoppingCondition $SC$
    Output: TestSuite $T$
 1: begin
 2:     $G \leftarrow$ Initial Random Population
 3:     repeat
 4:         $Z \leftarrow$ Elite($G$)
 5:         while $|Z| \neq |G|$ do $P_1, P_2 \leftarrow$ Selection($Z$)
 6:             if $XP$ then
 7:                 $O_1, O_2 \leftarrow$ Crossover($P_1, P_2$)
 8:             else
 9:                 $O_1, O_2 \leftarrow P_1, P_2$
10:             end if
11:             Mutate($O_1, O_2$)
12:             $f_P = \min(fitness(P_1), fitness(P_2))$
13:             $f_O = \min(fitness(O_1), fitness(O_2))$
14:             $l_P = length(P_1), length(P_2))$
15:             $l_O = length(O_1), length(O_2))$
16:             $T_B \leftarrow$ GetBestIndividual($G$)
17:             if $f_O < f_P$ or ($f_O = f_P$ and $l_O \leq l_P$) then
18:                 for $O$ in $O_1, O_2$ do
19:                     if $length(O) \leq 2 \times length(T_B)$ then
20:                         $Z \leftarrow Z \cup O$
21:                     else
22:                         $Z \leftarrow Z \cup \{P_1$ or $P_2\}$
23:                     end if
24:                 end for
25:             else
26:                 $Z \leftarrow Z \cup P_1, P_2$
27:             end if
28:         end while
29:         $G \leftarrow Z$
30:     until $fitness(GetBestIndividual(G)) = 0$ or $SC$ is met
31:     $T \leftarrow$ GetBestIndividual($G$)
32:     return $T$
33: end
```

2. improving the general testability (*e.g.,* convert the `String.equals()` into an edit distance to calculate the branch distance for strings).

Running EvoSuite requires two things: (i) a bytecode file (`.class`) containing a top-level class and (ii) the project's classpath — *i.e.* the list of paths that point to directories and/or JARs containing the classes referenced by the CUT. For example, considering the file `target/classes/example/MyCUT.class`, we would type the following command to launch the tool:

```
java -jar evosuite-1.0.6.jar -class example.MyCUT -projectCP target/classes
```

As soon as EvoSuite is launched, the generation for the class `MyCUT` begins and the produced JUnit test cases are collected in a class called `evosuite-tests/example/MyCUT_ESTest.java`. Each test method is accompanied by a set of assertions [30] that, however, must still be checked because they are based on the current behaviour of the CUT (*i.e.,* regression assertions), so if the latter has a defect the assertion will not catch it.

Algorithm 2 shows how the GA applied in EvoSuite works. Starting with a random population (Line 2), the evolution is performed until a solution that fulfills the selected coverage criterion is found, or the stopping condition is met (Line 30), *e.g.,* time or maximum number of evaluations. In each iteration of the evolution, a new generation is created and initialized with the best individuals (entire test suites) of the last generation (*i.e., elitism*, Line 4); then, this new generation is filled up with the best individuals produced by the genetic operators (*i.e.* selection, crossover and mutation) that also have to respect certain length constraints — namely, the new individuals have not to be longer than twice the length of the best individual (the entire loop of Line 5). The length of a test suite is given by the total number of statements among all the belonging test cases (Lines 14, 15 and 19).

The way the $fitness$ function (Lines 12, 13 and 30) works depend on the selected coverage criterion, for instance, if the chosen criterion is *branch coverage* then the formula is based on approach level and branch distance metrics.

Chapter 3

SIEGE: Automatic Exploitation of Known Vulnerabilities

This chapter introduces the technique SIEGE and explains it in details. Before diving into the algorithm design, we need to make some preliminary assumptions.

3.1 Preliminary Assumptions

We are focusing on a set of vulnerabilities affecting well-known OSS Java libraries and frameworks — *e.g.,* SPRING, JENKINS, TOMCAT, KAFKA — collected in the dataset provided by Ponta et al. [15]. This dataset contains more than 1000 known vulnerabilities, each described by:

– *CVE ID.*
– Project's *repository URL* (mostly GITHUB).
– *Fixing commit hash* signature.

Hence, the dataset is a set of triples ⟨CVE_ID, repository_url, fix_hash⟩, *e.g.,* ⟨CVE-2017-4971, `https://github.com/spring-projects/spring-webflow`, 57f2ccb⟩[15].

We chose to focus on what we believe to be the simplest form of vulnerability, that is the **vulnerable line**. Under this assumption (that we will call *the vulnerable line assumption* from this moment on), an exploit is any program execution

[15]The dataset provides full commit signatures, but here we preferred reported the equally valid short signature for the sake of brevity.

that covers a vulnerable line of an OSS library starting from the classes of a client project.

As anticipated, we chose to implement SIEGE as an extension for EvoSuite [14], mainly because it allows to:

– Add new coverage criteria;
– reuse well-validated meta-heuristics;
– rely on instrumentation and program analysis tools (*e.g.,* control flow graphs, call graphs and execution traces).
– define custom genetic operators (*i.e.,* selection, crossover and mutation);

It is possible to extend EvoSuite by adding some new classes — related to the new coverage criteria and genetic operators — and make small changes in others. For these reasons, EvoSuite is considered a fully-fledged GA framework.

3.2 SIEGE's Contribution and Workflow

As the acronym suggests, SIEGE is an approach for the automatic generation of exploit that relies on search techniques. SIEGE's novel contribution is twofold:

1. Providing a **new coverage criterion** that is a variant of line coverage specialized for reaching third-party library code from any client project's class.
2. Providing a **new fitness evaluation formula** that rewards the individuals whose executions better approach on fulfilling the criterion.

From a practical point of view, SIEGE consists in the addition of new classes in EvoSuite that represent both the new coverage criterion and fitness formula, namely:

– `VulnerabilityFitnessFactory`, that generates the coverage goals;
– `VulnerabilityTestFitness`, that represents a single coverage goal and computes the fitness score of a single test case;
– `VulnerabilitySuiteFitness`, that computers the fitness score of an entire test suite.

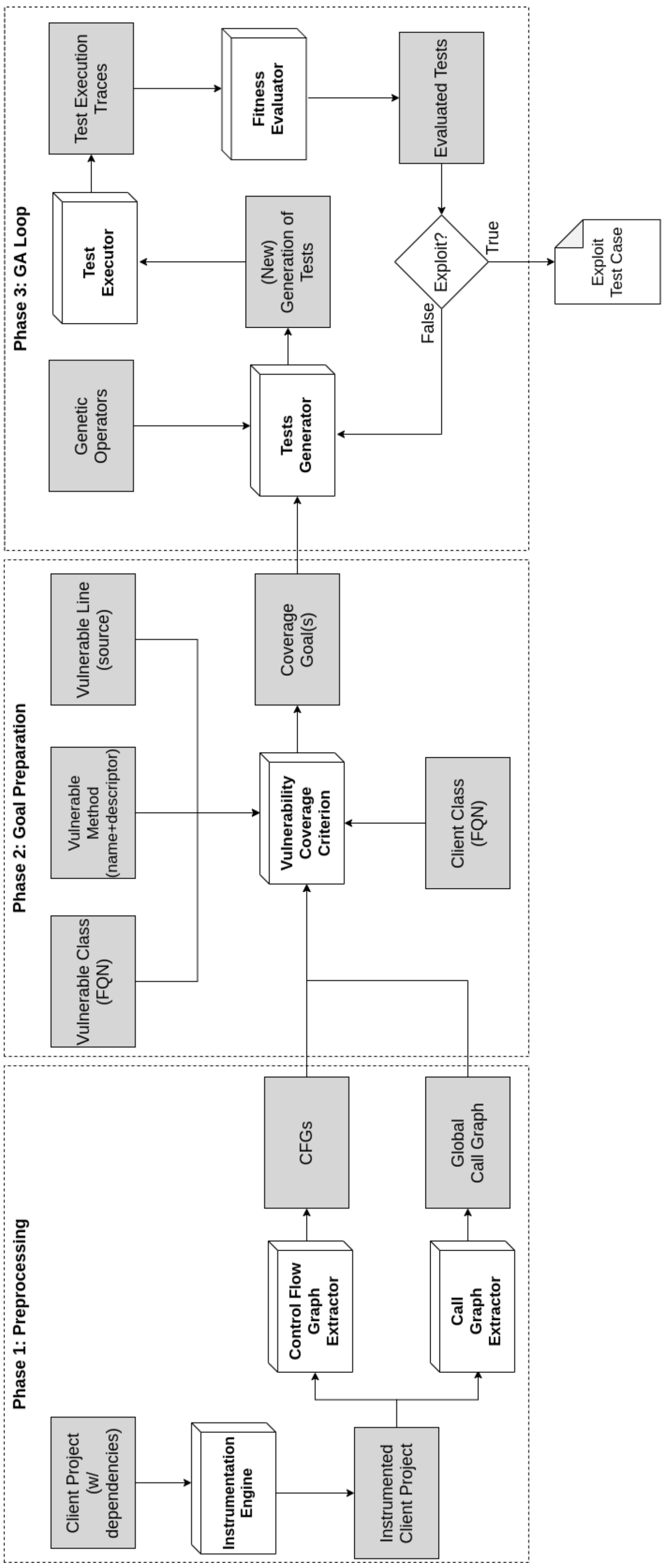


Figure 3.1: SIEGE's general workflow.

Figure 3.1 shows the general workflow of SIEGE, where three distinct phases can be observed: (i) *preprocessing*, (ii) *goal preparation* and (iii) *GA Loop*.

3.2.1 Preprocessing

SIEGE takes a client Java project in input along with its dependencies — that have to be provided through the project's classpath, optionally with the help of a dependency manager, such as Maven — and instruments the compiled bytecode. This phase has two main goals:

– convert the project into the form of a set of *control flow graphs*, and a *global call graph* that statically links them all by the means of method invocations;
– make the execution traces collectible during phase 3.

The extracted graphs will be used to build the path that connects the client classes to the vulnerable library constructs. For all these actions, SIEGE reuse EvoSuite's instrumentation and analysis tools.

3.2.2 Goal Preparation

Before running the GA, SIEGE prepares the coverage goals by taking:

– the *description* of a vulnerability, that uniquely indicates the position of the vulnerable line in a third-party library;
– the target client class name, that indicates the first class that will be invoked by an exploit.

Further details on goals' structure are explained in Section 3.3.2.

3.2.3 GA Loop

In this phase, SIEGE prepares the selected GA by giving the produced coverage goal(s) and the selected genetic operators, and starts its main loop. The GA starts evolving an initial population of candidate exploits and repeats the process until a concrete exploit is born or the stopping condition is met.

Each generated individual is executed against the client project (with all its classpath) and its trace (thanks to the previously done instrumentation), is evaluated against the coverage goal(s) to compute its fitness score and establish how close it is to become an exploit.

Further details on how this algorithm is designed in Section 3.3.

3.3 Genetic Algorithm Design

3.3.1 General Setup

In the context of this work we are not going to generate tests in the narrow sense of the term (*i.e.,* a program execution that verifies a certain expected behaviour) but 'exploits', so we safely disabled the EvoSuite's assertion generation engine.

Since we only need one exploit for a single vulnerability, we disabled the *whole suite* feature, in this manner the search will generate populations of single exploits instead of suites of exploits.

The entire algorithm is driven by the *Monotonic GA* meta-heuristic, an improvement of standard GA which, after mutating and evaluating each offspring, includes either the best offspring or the best parent in the next population (whereas the Standard GA includes both offsprings in the next population regardless of their fitness values) [31].

Currently, we chose not to focus on customizing the genetic operators, so we preferred to keep the default EvoSuite's choices, that are:

- *Rank Selection*, that creates an ordering of the individuals based on their fitness, and their survival likelihood is proportioned to their rank;
-
- *Single Point Crossover*, that crosses individuals' genetic structure by selecting a random split point;
- *Uniform Distribution Mutation*, that simply randomly mutate individuals by sampling from a uniform random distribution.

During the implementation we discovered some issues with the instrumentation engine of EvoSuite related to the *dynamic dispatch* (*i.e.,* the runtime resolution of the implementation of a polymorphic method). Namely, when an overridden method is called through the superclass reference the call is expected to contain the subclass' method but EvoSuite seems to incorrectly 'forget' adding it. In this way, many vulnerabilities that involve this mechanism, could not be considered. In the following we are considering only the vulnerabilities that avoid this issue.

Finally, we used time as stopping condition.

3.3.2 Coverage Goals

Any GA needs a proper modelling of the coverage goals, that constitute the set of all requirements to be fulfilled for finding the optimal solution. As assumed, we are relying on the concept of **covering a vulnerable line**, meaning that we need a way to uniquely describe the fact that an exploit covers (or not covers) the target line.

Similarly to ECLIPSE STEADY [10], if we consider the fixing commits of an OSS vulnerability we can infer the set of vulnerable constructs by looking at the modified line. Hence, a coverage goal is defined by a triple of strings ⟨VULNERABLE_CLASS, VULNERABLE_METHOD, VULNERABLE_LINE⟩, where:

– VULNERABLE_CLASS is the fully-qualified name (FQN) of the vulnerable class (*e.g.,* `hudson.tasks.Mailer`).
– VULNERABLE_METHOD is the name of the vulnerable method concatenated with its descriptor[16] (*e.g.,* `matches([B[B)Z`).
– VULNERABLE_LINE is the number of the vulnerable line in the source code[17].

VULNERABLE_CLASS and VULNERABLE_METHOD are used to determine the required *call context*, *i.e.,* the list of method calls from a client class to a library method to cover; VULNERABLE_LINE, instead, is used to retrieve the list of branches that 'dominate' the vulnerable line *i.e.,* the set of branches to be evaluated `true` to reach the block containing the line.

In other words, each coverage goal is composed of:

1. the TARGET CALL CONTEXT to reach the vulnerable method by a sequence of method calls;
2. the TARGET CONTROL NODES to reach the block of the vulnerable line;
3. the TARGET VULNERABLE LINE itself.

Any individual whose execution trace satisfies all these three requirements is an exploit for the given vulnerability.

3.3.3 Fitness Function

The computation of the fitness score of an individual is based on the *context similarity* and *approach level* measures:

[16]https://asm.ow2.io/
[17]Not to be confused with bytecode lines that may differ from source ones.

Definition 3.3.1. Given two call contexts A and B, the **context similarity** is the ratio of the number of method calls that A and B have in common out the total number of the longest call context between A and B.

An individual's execution trace covers various partial call contexts, so we need the *best context similarity* against the TARGET CALL CONTEXT T, *i.e.,* the maximum of all context similarities that can be computed against T.

Definition 3.3.2. Given an execution trace α and a branch b, the **approach level** is the number of control nodes that separate b and the branch $b^{'}$, that is the closest branch to b that is executed by α.

This definition can be extended to a list of branches (*i.e.,* control nodes), so the *best approach level* is the approach level of an individual's execution trace with respect to the 'innermost' control node, *i.e.,* the one that directly dominates the vulnerable line.

Algorithm 3 Fitness Computation

```
1: function computeFitness(CallContext $CC$, ControlNodes $CN$, Line $L$, ExecTrace $T$)
2:     $ctx \leftarrow$ ComputeBestContextSimilarity($T$, $CC$)
3:     if $ctx == 1$ then
4:         $al \leftarrow$ ComputeApproachLevel($T$, $CN$)
5:         if $al == 0$ then
6:             $cl \leftarrow$ GetClosestCoveredLine($T$, $L$)
7:             if $cl == L$ then
8:                 return 0
9:             else
10:                return $1 - (cl + 1)/(L - cl + 1)$
11:            end if
12:        else
13:            return $2 - (length(CN - al)/length(CN)$
14:        end if
15:    else
16:        return $3 - ctx$
17:    end if
18: end function
```

Algorithm 3 describes the fitness computation algorithm. Given an individual's execution trace T, the fitness function starts computing the best context similarity ctx with respect to the target call context CC and if there is a perfect match (Line 3) it proceeds to compute the approach level with respect to the target control nodes CN. If all control nodes are executed (Line 5) and the target line as well (Line 7) the fitness score is set to 0, flagging the goal as covered. In the cases where one of the requirements is not fulfilled, the fitness is set to a value greater than 0 (Lines 16, 13, 10).

3.4 Running Example

In this section we are going to see how SIEGE behaves in a common scenario.

```
302 public void parseCentralDirectoryFormat(final byte[] data, final int
        ↪offset, final int length) {
303   this.format = ZipShort.getValue(data, offset);
304   this.algId = EncryptionAlgorithm.getAlgorithmByCode(ZipShort.getValue
          ↪(data, offset + 2));
305   this.bitlen = ZipShort.getValue(data, offset + 4);
306   this.flags = ZipShort.getValue(data, offset + 6);
307   this.rcount = ZipLong.getValue(data, offset + 8);
308 
309   if (rcount > 0) {
310     this.hashAlg = HashAlgorithm.getAlgorithmByCode(ZipShort.getValue(
            ↪data, offset + 12));
311     this.hashSize = ZipShort.getValue(data, offset + 14);
312     // srlist... hashed public keys
313     for (int i = 0; i < this.rcount; i++){
314       for (int j = 0; j < this.hashSize; j++) {
315         // ZipUtil.signedByteToUnsignedInt(data[offset + 16 + (i * this.
                ↪hashSize) + j]));
316       }
317     }
318   }
319 }
```

Listing 3.1: Vulnerable method COMPRESS 1.15, related to CVE-2018-1324 that has a vulnerability on Line 313, depicted in yellow.

Let us consider the CVE-2018-1324 vulnerability that affects the version 1.15 of APACHE COMMONS COMPRESS — a library that offers an API for working with many compression algorithms. Listing 3.1 shows the affected method `parseCentralDirectoryFormat()` of class

```java
public class CompressCaller1 {
  private X0017_StrongEncryptionHeader seh;

  public CompressCaller1() {
    this.seh = new X0017_StrongEncryptionHeader();
  }
  public void call(byte[] data, int offset, int length) {
    this.seh.parseCentralDirectoryFormat(data, offset, length);
  }
}
```

Listing 3.2: Artificially crafted client class of the vulnerable method of CVE-2018-1324.

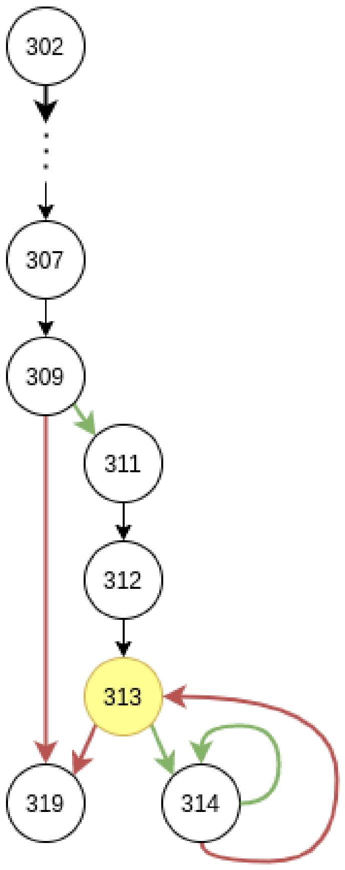


Figure 3.2: The control flow graph extracted from method `parseCentralDirectoryFormat()` of Listing 3.1. The green arrows represents the `true` branches, while the red ones represents the `false` branches. The yellow node (313) indicates the vulnerable line.

`X0017_StrongEncryptionHeader` and depicts the vulnerable line in yellow (inferred from the fixing commit 2A2F1DC), so the triple that describes

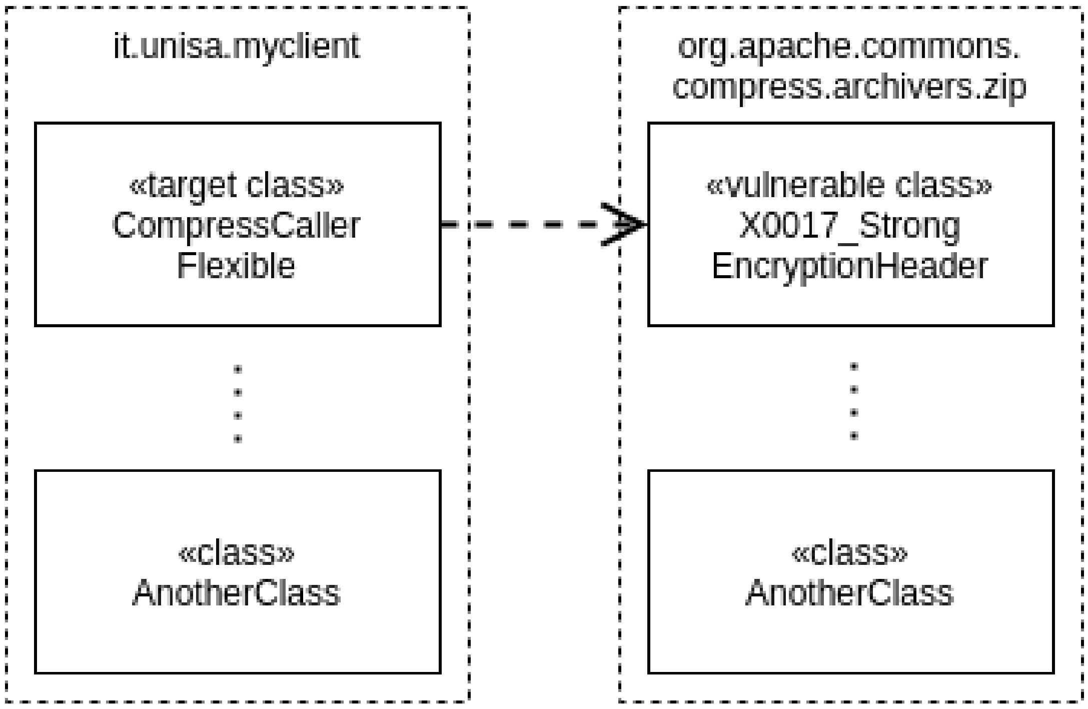

it.unisa.myclient
«target class» CompressCaller Flexible
«class» AnotherClass
org.apache.commons. compress.archivers.zip
«vulnerable class» X0017_Strong EncryptionHeader
«class» AnotherClass

Figure 3.3: The call graph extracted from class `CompressCallerFlexible` to `X0017_StrongEncryptionHeader`.

this vulnerability is ⟨`org.apache.commons.compress.archivers.zip.X0017_StrongEncryptionHeader`, `parseCentralDirectoryFormat([BII)V`, `313`⟩.

By reading the fixing commit message and the CVE-2018-1324 textual description, we see that this vulnerability may bring the program into an infinite loop state (*i.e.,* Denial of Service) if a certain ZIP file is given to `parseCentralDirectoryFormat()`; this happens due to an integer overflow on the loop index integer variable `i`, patched by simply changing its type to `long`.

We see that the vulnerable line is preceded by a set of method calls that may raise exceptions if certain conditions on `byte[] data` argument are not met. With simple program analysis, we can figure out that reaching the vulnerable line requires at least that:

1. `rcount` is greater than 0 (Line 309);
2. `byte[] data` is at least 16 bytes long to avoid an exception from method `ZipShort.getValue()` (Line 311);

The content of the `data` buffer influences the value of `rcount`, so fulfilling the first condition requires putting certain values into `data`; however, we do not know which are these values, so we leave the decision to the generation of SIEGE's GA. In other words, every generated individual that calls `parseCentralDirectoryFormat()` and satisfies these two conditions should be a concrete exploit[18].

Let us provide an artificially crafted client class, called `CompressCallerFlexible`, shown in Listing 3.1. This class calls `parseCentralDirectoryFormat()` in method `call()` (Line 7) with variable actual arguments, depending on the values generated by the GA.

The preprocessing phase extracts the control flow graph from the vulnerable method `parseCentralDirectoryFormat()` (Figure 3.2) and the call graph from the client class `CompressCallerFlexible` to the target vulnerable class `X0017_StrongEncryptionHeader` (Figure 3.3). These two structures are used for building the following coverage goal:

- TARGET CALL CONTEXT spans across two methods:
    1. `it.unisa.myclient.CompressCaller1:call([BII)V`, of the client project;
    2. `org.apache.commons.compress.archivers.zip.X0017_ StrongEncryptionHeader:parseCentralDirectoryFormat([BII)V` of the target library.
- TARGET CONTROL NODES are:
    1. `CD I50 Branch 1 IFLE L309 - FALSE`, meaning that the bytecode conditional expression related to the source line 309 have to be evaluated `false`[19].
- TARGET VULNERABLE LINE is 313.

The outcome of SIEGE strongly depends on the given search budget. For instance, if we run it with 5 seconds of budget, the GA is not able to properly evolve the population.

Listing 3.3 shows one of the best individuals from the first generation. We can clearly see that without any suggestion from previous iterations, SIEGE generates an individual with random values (along with some useless statements).

[18]Actually, the infinite loop does not happen if `rcount` is not high enough (*i.e.,* over the maximum for 32-bit signed integers, namely $2,147,483,647$) to cause the `i` variable to overflow.

[19]It is `false` instead of `true` because bytecode flips the conditionals.

```java
CompressCaller1 compressCaller1_0 = new CompressCaller1();
byte[] byteArray0 = new byte[2];
byte byte0 = (byte)0;
byteArray0[0] = byte0;
byte byte0 = (byte) (-8);
byteArray0[1] = byte0;
int int0 = (-2552);
compressCaller1_0.call(byteArray0, byte0, int0);
```

Listing 3.3: One of the best individuals from the first generation obtained from an execution of SIEGE for CVE-2018-1324 with insufficient search budget with fitness score of 2.

```java
CompressCaller1 compressCaller1_0 = new CompressCaller1();
byte[] byteArray0 = new byte[12];
int int0 = 1220;
compressCaller1_0.call(byteArray0, byteArray0[1], int0);
```

Listing 3.4: Imperfect individual obtained from an execution of SIEGE for CVE-2018-1324 with insufficient search budget. It stops the line before the vulnerable line, making it a quasi-exploit.

```java
CompressCaller1 compressCaller1_0 = new CompressCaller1();
byte[] byteArray0 = new byte[19];
byteArray0[0] = (byte)3;
byteArray0[14] = (byte)2;
byteArray0[2] = (byte)2;
byteArray0[3] = (byte) (-1);
byteArray0[4] = (byte) (-1);
compressCaller1_0.call(byteArray0, (byte)3, 2);
```

Listing 3.5: Perfect individual obtained from an execution of SIEGE for CVE-2018-1324 with sufficient search budget. It covers the vulnerable line, making it an exploit.

Listing 3.4 shows an 'imperfect' (*i.e.,* has not fully covered the entire coverage goal) best individual from last generation (the 72th), with fitness score of 0.364. We can see that it passes a 12 bytes long `data`, so failing at preparing the right content and raising an exception at the line before the vulnerable one (Line 311).

On the other hand, with a sufficiently large search budget, such as 60 seconds, the GA has enough time to produce a perfect individual (*i.e.,* with fitness score of 0) within 656 generations. Listing 3.5 shows a perfect individual which correctly passes a 19 bytes long `data` with the proper content to cover the vulnerable line.

We can see that if we tweaked some GA's settings we could achieve highly different results. In our empirical validation in Chapter 4 we are going to explore this aspect by considering different vulnerabilities in terms of complexity.

Chapter 4

Empirical Validation

This section shows the first empirical validation of SIEGE. Being an initial implementation, we decided to validate it on the heels of the running example seen in Section 3.4. The results are collected and discussed at the end of the section, along with the threats to the validity of the study.

4.1 Research Goals and Questions

Our goal was to start from a preliminary investigation that could assess the feasibility of the idea behind SIEGE. For this reason, we decided not to conduct a full empirical study.

The first research question is about the effectiveness of SIEGE in the exploitation of a set of known OSS vulnerabilities:

RQ$_1$. *Does* SIEGE *succeed in the generation of exploits for a set of OSS vulnerabilities?*

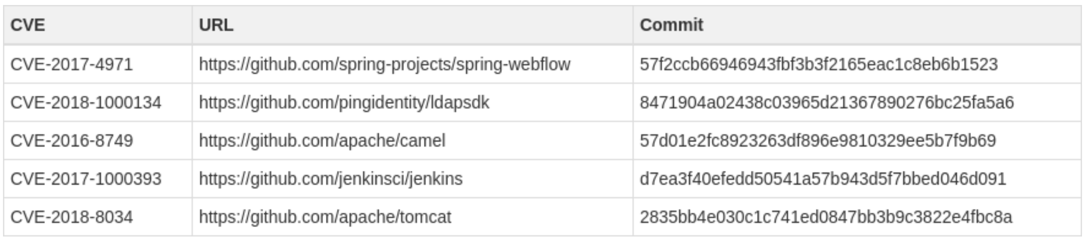
We expect that not all vulnerabilities are the same: some may be easier to exploit, while others may require certain objects to be put in a very precise state in order to cover the vulnerable line. The GA of SIEGE, probably, will not be able to correctly generate these objects if the allotted search budget is not enough, so we are going to execute it with different budgets to see how it reacts under the various situations. We say that the vulnerabilities that SIEGE is not able to exploit under all different settings will be classified as *complex*.

We suspect that SIEGE's performance is influenced by how a client class uses the vulnerable constructs. For this reason, we ask this second research question:

RQ$_2$. *Is the exploitability affected by how client classes use the vulnerable constructs?*

4.2 Dataset Construction

We selected a restricted set of known OSS vulnerabilities from the dataset of Ponta et al. [15], containing 1282 known vulnerabilities affecting hundreds of Java OSS libraries and frameworks. Figure 4.1 shows its first five rows.



CVE	URL	Commit
CVE-2017-4971	https://github.com/spring-projects/spring-webflow	57f2ccb66946943fbf3b3f2165eac1c8eb6b1523
CVE-2018-1000134	https://github.com/pingidentity/ldapsdk	8471904a02438c03965d21367890276bc25fa5a6
CVE-2016-8749	https://github.com/apache/camel	57d01e2fc8923263df896e9810329ee5b7f9b69
CVE-2017-1000393	https://github.com/jenkinsci/jenkins	d7ea3f40efedd50541a57b943d5f7bbed046d091
CVE-2018-8034	https://github.com/apache/tomcat	2835bb4e030c1c741ed0847bb3b9c3822e4fbc8a

Figure 4.1: Head of the dataset of Ponta et al.

The selection was supported by a PyDriller [32] script, that quickly let us filter-in the vulnerabilities that met these criteria:

1. the fixing commit involves one and only one Java class;
2. the fixing commit involves a single line (to respect the vulnerable line assumption).

In addition, the filtered-in vulnerabilities were manually inspected — by comparing their fixing commits with their last vulnerable version — for two main reasons:

1. to establish whether they were not impacted by the instrumentation issue of EvoSuite (addressed in Section 3.3.1);
2. to extract the vulnerable constructs (namely, the class, the method and the line) needed by SIEGE to build the coverage goals.

The inspection involved 28 different vulnerabilities and costed a total of 5 man-hours. Finally, we have chosen 11 out 28 vulnerabilities (Table 4.1) to be part of the validation, belonging to 11 different projects to have greater diversity.

Vulnerability	**Library**	**Version**	**Description**
CVE-2018-1324	Apache Commons Compress	1.15	DoS when a certain input (zip file) is provided
CVE-2011-1582	Apache Tomcat	7.0.12	Using a wrong classloader to bypass access restrictions
CVE-2014-9970	Jasypt	1.9.1	Exposition to Timing Attacks by using a linear time hashes comparison function
CVE-2018-1000067	Jenkins	2.89.3	Improper access restriction for a restricted functionality
CVE-2017-1000390	Tikal Multijob Plugin	1.26	Improper access restriction for a restricted functionality
CVE-2016-3092	Apache Commons File Upload	1.3.1	DoS due to certain sufficiently large buffers
CVE-2011-1498	Apache HttpClient	4.1	Exposing sensitive information in logs
ZEPPELIN-2769	Apache Zeppelin	0.6.0	Exposure to SQL Injection
CVE-2018-17194	Apache Nifi	1.7.1	Time waste with a certain DELETE HTTP requests with non-empty body
CVE-2018-8718	Mailer Plugin	1.20	Exposition to CSRF for a certain functionality
PRIMEFACES-1194	Primefaces	6.1	Exposition to XSS due to a lack of escaping of a user input

Table 4.1: The set of 11 vulnerabilities considered for the validation. All these versions are available at Maven Central Repository.

4.3 Research Methodology and Data Analysis

```java
public class JasyptCaller1 {
  private StandardByteDigester standardByteDigester;
  public JasyptCaller1(int bytes) {
    this.standardByteDigester = new StandardByteDigester();
    this.standardByteDigester.setSaltSizeBytes(bytes);
  }
  public void call(byte[] message, byte[] digest) {
    this.standardByteDigester.matches(message, digest);
  }
}

public class JasyptCaller2 {
  private StandardByteDigester standardByteDigester;
  public JasyptCaller2(StandardByteDigester standardByteDigester) {
    this.standardByteDigester = standardByteDigester;
  }
  public void call(byte[] message, byte[] digest) {
    this.standardByteDigester.matches(message, digest);
  }
}
```

Listing 4.1: Two artificial client classes that call the vulnerable method of JASYPT 1.9.1, related to CVE-2014-9970. Their difference stands in their constructors, where the first one creates the instance of `StandardByteDigester` and use the `bytes` value generated by SIEGE, while the second one directly receive an instance of `StandardByteDigester` from SIEGE.

To answer **RQ**$_1$ we run SIEGE on the selected vulnerabilities 3 times in order to remove the effects caused by the random nature of the GA, and set the size of the population (*i.e.*, the maximum number of evolving individuals) to 100 because we were not interested in investigating this factor. The results are express in terms of median fitness scores of the best individuals and the median number of the elapsed generations over the 3 runs.

We can fully answer **RQ**$_1$ if, and only if, all 11 vulnerabilities are exploited by SIEGE, otherwise we will provide a partial answer and analyze the causes of the failure.

To answer **RQ**$_2$ we required different client classes that call the selected vulnerabilities in various ways. Since finding a proper set of real JAVA projects that include and use these 11 libraries would have been tedious and unsuccessful, we decided to develop some *artificial client classes* that act in place of client projects and explicitly call the vulnerable constructs. These classes were created by considering the control flow of the vulnerable method and the required conditions to reach the vulnerable line. In particular, we developed two sets of artificial classes: the first ones are in charge of (i) instantiating the vulnerable class with fixed values and (ii) calling the vulnerable method with the values generated by the GA (similarly to what we have seen in Section 3.4); the second ones, instead, leave everything to the GA's generation, including the values needed by one of the vulnerable class' constructors. In other words, the formers should be well driven towards the vulnerable line, while the latters should give the GA more degrees of freedom, as shown in Listing 4.1. The two sets differ only for CVE-2016-3092, where the vulnerability directly affects one of the constructors, and since we could not let the GA instantiate the vulnerable class freely, we decided to give this vulnerability just one client, belonging to the first set. SIEGE will be run a total of 6 times, 3 times per set.

We can positively answer $\mathbf{RQ}_2$ if, and only if, we see that the two sets of clients score different results (in terms of fitness scores of the best individual). On the contrary, if we see comparable results we will provide a negative answer.

4.4 Results

Set #1	**Budget (s)**			
Vulnerable Library	5	15	30	60
Compress	0.364 (37)	0.364 (164)	0.000 (336)	0.000 (285)
Tomcat	0.000 (1)	0.000 (1)	0.000 (1)	0.000 (1)
Jasypt	0.000 (1)	0.000 (1)	0.000 (1)	0.000 (1)
Jenkins	0.000 (1)	0.000 (1)	0.000 (1)	0.000 (1)
Multijob	0.000 (1)	0.000 (1)	0.000 (1)	0.000 (1)
File Upload	0.000 (1)	0.000 (1)	0.000 (1)	0.000 (1)
HttpClient	0.000 (1)	0.000 (1)	0.000 (1)	0.000 (1)
Zeppelin	0.000 (1)	0.000 (1)	0.000 (1)	0.000 (1)
Nifi	3.000 (43)	3.000 (225)	3.000 (499)	3.000 (1079)
Mailer	3.000 (82)	3.000 (241)	3.000 (477)	3.000 (998)
Primefaces	2.000 (34)	2.000 (111)	2.000 (226)	2.000 (454)

Table 4.2: The results of the validation on the first set of artificial client classes. The cells report the median fitness score of the best individual and the median number of elapsed generations in parentheses over 3 runs.

Set #2	**Budget (s)**			
Vulnerable Library	5	15	30	60
Compress	2.000 (19)	0.364 (29)	0.000 (30)	0.000 (30)
Tomcat	0.000 (1)	0.000 (1)	0.000 (1)	0.000 (1)
Jasypt	0.000 (2)	0.000 (2)	0.000 (2)	0.000 (2)
Jenkins	0.000 (1)	0.000 (1)	0.000 (1)	0.000 (1)
Multijob	2.500 (33)	2.500 (86)	2.500 (168)	2.500 (307)
HttpClient	0.000 (1)	0.000 (1)	0.000 (1)	0.000 (1)
Zeppelin	0.000 (1)	0.000 (1)	0.000 (1)	0.000 (1)
Nifi	0.000 (1)	0.000 (1)	0.000 (1)	0.000 (1)
Mailer	2.500 (18)	2.500 (61)	2.500 (129)	2.500 (259)
Primefaces	2.000 (20)	2.000 (57)	2.000 (95)	2.000 (196)

Table 4.3: The results of the validation on the second set of artificial client classes. The cells report the median fitness score of the best individual and the median number of elapsed generations in parentheses over three runs.

The results of the empirical validation on the first and second set of artificial clients are collected in Tables 4.2 and 4.3, respectively.

On the first set of artificial clients, 7 out 11 (63.636%) vulnerabilities were correctly exploited with any given search budget and within at most 2 generations; among the remaining 4 vulnerabilities, only Compress required a higher budget to be exploited (namely, 30 and 60 seconds). Unfortunately, for Nifi and Mailer, SIEGE scored the worst possible fitness value of 3, and 2 for Primefaces.

The whole story is very similar for the second set, but there are some interesting differences with respect to the first set to highlight:

– The overall number of elapsed generations is lower. This might be caused by the fact that the clients of the second set delegate the vulnerable class instantiation to the GA's generation — in addition to the actual arguments to call the vulnerable method — causing SIEGE to generate many more values in comparison.
– The generation for Compress became slightly harder with low budget;
– The generation failed for Multijob. SIEGE might have found more troubles in the object creation than in the vulnerable method call;
– The generation for Nifi became much easier, succeeding even with low budget. This hints an incorrect development of the first client class that hinders

```
public class StandardWrapper extends ContainerBase implements
    ↪ServletConfig, Wrapper, NotificationEmitter {
  public void servletSecurityAnnotationScan() throws ServletException {
    if (getServlet() == null) {
      Class<?> clazz = null;
      try {
         clazz = getParentClassLoader().loadClass(getServletClass());
         processServletSecurityAnnotation(clazz);
      } catch(ClassNotFoundException e) {
        // Safe to ignore. No class means no annotations to process
      }
    } else {
      if (servletSecurityAnnotationScanRequired) {
        processServletSecurityAnnotation(getServlet().getClass());
      }
    }
  }
}

public class TomcatCaller1 {
  private StandardWrapper standardWrapper;
  public TomcatCaller1(Servlet servlet, boolean b) {
    this.standardWrapper = new StandardWrapper();
    standardWrapper.setServlet(servlet);
    standardWrapper.setServletSecurityAnnotationScanRequired(b);
  }
  public void call() throws ServletException {
    this.standardWrapper.servletSecurityAnnotationScan();
  }
}
```

Listing 4.2: The vulnerable method of Tomcat 7.0.12 related to CVE-2011-1528 (above) and its artificial client class from the first set (below). The vulnerable line is depicted in yellow.

```
TomcatCaller1 tomcatCaller1_0 = new TomcatCaller1((Servlet) null, false);
tomcatCaller1_0.call();
```

Listing 4.3: The best individual obtained from an execution of SIEGE for CVE-2011-1528 with a search budget of 5 seconds. It is a valid exploit.

the GA;

– The generation for Mailer and Primefaces remained similar, with just a slight improvement for Mailer.

Listings 4.2 and 4.3 show the vulnerable method of Tomcat 7.0.12, its client class and the easily generated exploit. We can see that a `null` value as `Servlet`

```java
public class MultiJobResumeBuild implements RunAction2, StaplerProxy {
  public Object getTarget() {
    Jenkins.getInstance().checkPermission(Jenkins.ADMINISTER);
    return this;
  }
}

public class JenkinsMultijobCaller2 {
  private MultiJobResumeBuild build;
  public JenkinsMultijobCaller2(MultiJobResumeBuild build) {
    this.build = build;
  }
  public void call() {
    this.build.getTarget();
  }
}
```

Listing 4.4: The vulnerable method of Multijob 1.26 related to CVE-2017-1000067 (above) and its artificial client class from the second set (below). The vulnerable line is depicted in yellow.

```java
MultiJobResumeBuild multiJobResumeBuild0 = null;
JenkinsMultijobCaller2 jenkinsMultijobCaller2_0 = new
    ↪JenkinsMultijobCaller2(multiJobResumeBuild0);
jenkinsMultijobCaller2_0.call();
```

Listing 4.5: The best individual obtained from an execution of SIEGE for CVE-2017-1000067 with a search budget of 60 seconds. It is not an exploit, since its fitness is 2.5.

argument is enough for reaching the vulnerable. On the contrary, Listings 4.4 and 4.5 show the vulnerable method of Multijob 1.26, its client class and the best individual, that is far from being an exploit. By analysing the vulnerable class, the required argument for the constructor should have been of class `Run<MavenModule, MavenBuild>`.

4.5 Discussion

The empirical validation on the two sets of artificial clients let us discover some interesting insights on the selected vulnerabilities and on the behaviour of SIEGE; moreover, it allowed us to find some points of improvement and possible extensions.

Not all 11 vulnerabilities were successfully exploited, so we can only partially answer **RQ**$_1$. However, as there exist hard, or even infeasible, branches in ATCG, so there exist hard-to-exploit vulnerabilities, thus we expected a finding like this.

We can see that the allotted search budget has an important impact in the exploit generation, but the strongest effects come from (i) the vulnerability itself and (ii) the client class. We deduce that there are some vulnerabilities whose exploitability strongly depends on how the client call them, while there are others that are intricately complex, requiring a very precise setup by the client. This finding fully answers $\mathbf{RQ}_2$.

This investigation shows promising results despite the many constraints and simplifications we have imposed, and that we hope to remove as soon as possible. One of the next steps is to reimplement the technique in another way since EvoSuite is unsuitable for our needs due to the numerous limitations related to its instrumentation engine.

As for future enhancements, we think that the exploitation of complex vulnerabilities could be improved in various ways, such as:

– improving the entire GA, such as designing dedicated meta-heuristics, genetic operators, etc.;
– providing a novel definition of the concept of vulnerability — that goes beyond the simple vulnerable line — to capture the different shades of the defect.

Moreover, the effort that SIEGE puts during the exploit generation could be used as a **complexity measure** of the vulnerabilities; for instance, we could say that, according to our results, CVE-2011-1582 (Tomcat 7.0.12) is less complex than CVE-2018-1324 (Compress 1.15), that in turn is less complex CVE-2018-8718 (Mailer 1.20). This complexity measure will address one of the open issues related to the vulnerability assessment.

As said, the way a client class uses the vulnerable constructs highly affect the effectiveness of the GA. In particular, if a client does not build the needed complex objects on its own, the GA is forced to 'struggle' for generating the proper values, spending less time in the effective exploitation. Since we do not know how realistic our artificial clients are, we highlight the urgency of switching to **real client projects** to conduct a **large scale empirical study**.

Given the complexity measure and other elements — such as the API stability [21] and popularity [22, 23] — we could enhance SIEGE with a novel **library migration process** that would suggest if the update is urgent (*i.e.,* the vulnerability is easy to exploit) and which is the best available non-vulnerable version that minimize the risk of the update.

The general idea of SIEGE brought out the opportunity to extend the technique to the **automatic integration test generation**, hinted by the fact that the generated exploits span across multiple classes, just like integration tests (the only difference stands in the absence of assertions).

4.6 Threats to Validity

This empirical validation cannot be considered a fully-fledged empirical study, but rather a preliminary investigation, and it brings a number of threats to validity.

4.6.1 Threats to Construct Validity

Threats to *construct* validity are concerned with the relation between theory and observation.

We introduced the concept of complexity of vulnerabilities to explain why SIEGE sometimes have difficulty to generate exploits for certain vulnerabilities. We have given an intuitive definition of complexity that is based on the SIEGE's behaviour and we are aware of the fact that it should be based on a known measure that, however, is absent in literature at the best of our knowledge.

We selected the 11 vulnerabilities through manual inspection, without using any elements of randomness, so we might have picked 'too good' cases; this choice was motivated by the fact that we wanted to avoid the instrumentation issue of EvoSuite and to honor the vulnerable line assumption.

4.6.2 Threats to Internal Validity

Threats to *internal* validity concern factors that might have influenced the causal relationship between treatment and outcome.

There are several factors — such as the chosen genetic operators, the meta-heuristic and other GA parameters — that may have a significant impact on the generation of exploits. We have deliberately not considered these factors to reduce the scope of this preliminary investigation and we need to take them into account in future studies.

4.6.3 Threats to Conclusion Validity

Threats to the *conclusion* validity are concerned with issues that affect the ability to draw the correct conclusion about relations between treatment and outcome.

We launched SIEGE on a total of 21 client classes, divided in two sets based on how they call the vulnerable class. These artificial classes were developed by considering the existence of the related vulnerability and how the latter could be easily reached by the candidate exploits. Despite we were aware of this bias, we thought it was a needed step to evaluate the feasibility of the methodology before switching on real client projects.

4.6.4 Threats to External Validity

Threats to *external* validity are conditions that limit the ability to generalize the results to a broader environment.

As said, our validation is just a preliminary investigation, whose goal was to understand the feasibility of the methodology and to explore the various forms of vulnerabilities. Currently, it is hard to generalize these results to other forms outside the vulnerable line, so we need another empirical study before.

Chapter 5

Conclusion

We presented SIEGE, a novel search-based automatic exploit generation technique for JAVA projects which include OSS library versions with known vulnerabilities. We validated it on a set of 11 vulnerabilities of famous JAVA OSS libraries with two sets of client classes and different values of search budget, achieving successful exploitation for over 60% of vulnerabilities and generally within 5 seconds.

We learnt two important lessons:

1. the idea behind SIEGE is valid and shows promising results, but we need to overcome its flaws and limitations, mainly the vulnerable line assumption;
2. there exist vulnerabilities that are more complex than others in terms of exploitability, and they are quietly influenced by the way client classes use them.

We experienced an important issue of EVOSUITE related to the instrumentation engine that hindered our empirical validation; since we image it will cost us a lot in the future, we plan to reimplement SIEGE either onto another framework or from scratch.

As for our future agenda, we are going to (i) overcome the vulnerable line assumption and work on a general, and more comprehensive, definition of vulnerabilities and (ii) run a large scale empirical study that involve a high number of OSS vulnerabilities and real client projects.

Ringraziamenti

Questa tesi marca la fine di un lungo percorso formativo, non solo come studente ma, specialmente, come persona.

Questi due anni di Laurea Magistrale hanno continuato un lavoro di crescita già avviato in quell'ormai lontano Settembre 2015, dove un piccolo ragazzino sbarbato iniziava la sua prima lezione di *Architettura degli Elaboratori*, un esame che mi ha fatto capire che ero davvero pronto per lo studio universitario e che avrei potuto ottenere grandi risultati se solo mi fossi impegnato costantemente. Nei cinque anni successivi le conferme alle mie capacità non sono mai mancate, superando, sessione dopo sessione, ogni ostacolo fino all'ultimo esame del Giugno 2020: *Software Dependability*, che annovero tra i più soddisfacenti di sempre.

Ma se da un lato le mie conoscenze aumentavano, dall'altro cresceva in me la consapevolezza che ciò che sapevo era solo una minima parte di quello che il mondo era in grado di offrirmi; e quando ho iniziato a pensare che ciò che facevo non fosse realmente così eccezionale, ho avuto tante persone al mio fianco che, direttamente o indirettamente, mi hanno ricordato che bisogna saper godere dei propri successi e avere sempre viva la voglia di imparare cose nuove.

Voglio iniziare col ringraziare le persone che hanno direttamente contribuito a questo lavoro.

Ai professori Fabio Palomba e Andrea De Lucia va il mio *grazie*. Avete creduto in me anche quando non ne capivo le ragioni e avete sempre avuto una grande pazienza, nonostante le mie continue distrazioni e preoccupazioni. A questo *grazie* aggiungo i dottori Fabiano Pecorelli, Dario Di Nucci e Antonino Sabetta per il loro supporto e consiglio.

Ho potuto apprendere molto dai vari corsi, ma l'insegnamento vero, si sa, lo danno le persone: i miei colleghi universitari, compagni di molte avventure e protagonisti di mille aneddoti.

Alla triade, Sara, Stefano e Umberto, va il mio *grazie*. Siete voi l'inizio del mio intero percorso accademico.

Ai magici, FLAVIO e MANUEL, va il mio *grazie.* I progetti senza di voi sarebbero andati fuori budget e fuori schedule.

“Chi trova un amico trova un tesoro” si dice, ma sono dell’idea che l’amicizia non si trovi già pronta ma vada costruita con le azioni di ogni giorno.

Ai maestri, ALFREDO, FEDERICO, FRANCESCO, SIMONE e VINCENZO, va il mio *grazie.* L’amicizia di coloro che ci sono sempre ha un valore inquantificabile.

“La famiglia non si sceglie”, sarà vero, ma questo non significa che non si possa essere grati ogni giorno del posto in cui è cresciuto.

Ai *boomer*, CARMELA e GIANNI, va il mio *grazie.* Non mi avete fatto mancare mai niente, e adesso tocca a me “ripagarvi” ogni cosa.

Al guardiano, FRANCESCO, va il mio *grazie.* In questi ultimi anni abbiamo riscoperto il valore dell’essere fratelli e la bellezza dell’essere fieri l’uno dell’altro.

L’amore è davvero inspiegabile. Ti fa dire cose e compiere scelte che ai più possono sembrare assurde, ma che per te e per quella persona speciale hanno un senso vero e pieno.

All’amore della mia vita, TANIA, va il mio *grazie.* Mi hai reso l’uomo che sono, insegnandomi che sono speciale e che insieme si vince sempre su tutto. *“You make my heart feel like it’s summer when the rain is pouring down”.*

Dalla vita di un un uomo non si escludere ciò che lo rende tale. A DIO va il mio *grazie* per avermi fatto così e per avermi insegnato ad amarmi per quello che sono, nonostante i miei difetti.

Emanuele

Bibliography

[1] Pasquale Salza et al. “Third-party libraries in mobile apps”. In: *Empirical Software Engineering* 25 (2019), pp. 2341–2377.

[2] J. Bosch. “From software product lines to software ecosystems”. In: *SPLC*. 2009.

[3] Mike Pittenger. *Open source security analysis: The state of open source security in commercial applications*. Black Duck Software, Tech. Rep. 2020. URL: `https://www.vojtechruzicka.com/bf4dd32d5823c258c319cced38727dce/OSSAReport.pdf` (visited on 07/12/2020).

[4] Joseph Hejderup. “In Dependencies We Trust: How vulnerable are dependencies in software modules?” PhD thesis. May 2015.

[5] Alexandre Decan, Tom Mens, and Eleni Constantinou. “On the Impact of Security Vulnerabilities in the Npm Package Dependency Network”. In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR ’18. Gothenburg, Sweden: Association for Computing Machinery, 2018, 181–191. ISBN: 9781450357166. DOI: `10.1145/3196398.3196401`. URL: `https://doi.org/10.1145/3196398.3196401`.

[6] OWASP. *OWASP Dependency Check*. 2020. URL: `https://owasp.org/www-project-dependency-check/` (visited on 07/12/2020).

[7] Whitesource Software. *Whitesource*. 2020. URL: `https://www.whitesourcesoftware.com/oss_security_vulnerabilities/` (visited on 07/12/2020).

[8] Spot Bugs. *Spot Bugs*. 2020. URL: `https://spotbugs.github.io/` (visited on 07/12/2020).

[9] SonarQube. *Static Application Security Testing.* 2020. URL: `https://www.sonarqube.org/features/security/` (visited on 07/12/2020).

[10] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. "Beyond Metadata: Code-centric and Usage-based Analysis of Known Vulnerabilities in Open-source Software". In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME).* 2018.

[11] Raula Kula et al. "Do developers update their library dependencies? An Empirical Study on the Impact of Security Advisories on Library Migration". In: *Empirical Software Engineering* (May 2017), pp. 1–34. DOI: `10.1007/s10664-017-9521-5`.

[12] D. Landman, A. Serebrenik, and J. J. Vinju. "Challenges for Static Analysis of Java Reflection - Literature Review and Empirical Study". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* 2017, pp. 507–518. DOI: `10.1109/ICSE.2017.53`.

[13] Sadeeq Jan et al. "Search-based Multi-Vulnerability Testing of XML Injections in Web Applications". In: *Empirical Software Engineering* (Apr. 2019). DOI: `10.1007/s10664-019-09707-8`.

[14] Gordon Fraser and Andrea Arcuri. "Whole Test Suite Generation". In: *IEEE Transactions on Software Engineering* 39.2 (2013), pp. 276–291. ISSN: 2326-3881. DOI: `10.1109/TSE.2012.14`.

[15] Serena Ponta et al. "A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software". In: Feb. 2019.

[16] Gabriele Bavota et al. "How the Apache Community Upgrades Dependencies: An Evolutionary Study". In: *Empirical Softw. Engg.* 20.5 (Oct. 2015), 1275–1317. ISSN: 1382-3256. DOI: `10.1007/s10664-014-9325-9`. URL: `https://doi.org/10.1007/s10664-014-9325-9`.

[17] Danny Dig and Ralph Johnson. "How Do APIs Evolve? A Story of Refactoring: Research Articles". In: *J. Softw. Maint. Evol.* 18.2 (Mar. 2006), 83–107. ISSN: 1532-060X.

[18] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Boston, MA, USA: Addison-Wesley, 1999. ISBN: 0-201-48567-2.

[19] Emanuele Iannone et al. "The Secret Life of Software Vulnerabilities - Technical Report". URL: https://emaiannone.github.io/download/other/secret_life.pdf.

[20] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. "When do changes induce fixes?" In: vol. 30. July 2005. DOI: 10.1145/1082983.1083147.

[21] S. Raemaekers, A. van Deursen, and J. Visser. "Measuring software library stability through historical version analysis". In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 2012, pp. 378–387.

[22] Andre Hora and Marco Valente. "apiwave: Keeping Track of API Popularity and Migration". In: Sept. 2015, pp. 1–3. DOI: 10.1109/ICSM.2015.7332478.

[23] Yana Momchilova Mileva et al. "Mining Trends of Library Usage". In: *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*. IWPSE-Evol '09. Amsterdam, The Netherlands: Association for Computing Machinery, 2009, 57–62. ISBN: 9781605586786. DOI: 10.1145/1595808.1595821. URL: https://doi.org/10.1145/1595808.1595821.

[24] Rodrigo Zapata et al. "Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages". In: Sept. 2018, pp. 559–563. DOI: 10.1109/ICSME.2018.00067.

[25] Saswat Anand et al. "An Orchestrated Survey of Methodologies for Automated Software Test Case Generation". In: *J. Syst. Softw.* 86.8 (2013), 1978–2001. ISSN: 0164-1212. DOI: 10.1016/j.jss.2013.02.061. URL: https://doi.org/10.1016/j.jss.2013.02.061.

[26] Kalyanmoy Deb. "Multi-Objective Optimization". In: *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Ed. by Edmund K. Burke and Graham Kendall. Boston, MA: Springer US, 2005, pp. 273–316. ISBN: 978-0-387-28356-2. DOI: 10.1007/0-387-28356-0_10. URL: https://doi.org/10.1007/0-387-28356-0_10.

[27] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. "Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets". In: *IEEE Transactions on Software Engineering* 44 (Feb. 2018), pp. 122–158. DOI: 10.1109/TSE.2017.2663435.

[28] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. "Reformulating Branch Coverage as a Many-Objective Optimization Problem". In: Apr. 2015. DOI: `10.1109/ICST.2015.7102604`.

[29] Andrea Arcuri. "Many Independent Objective (MIO) Algorithm for Test Suite Generation". In: *Lecture Notes in Computer Science* (2017), 3–17. ISSN: 1611-3349. DOI: `10.1007/978-3-319-66299-2_1`. URL: `http://dx.doi.org/10.1007/978-3-319-66299-2_1`.

[30] G. Fraser and A. Zeller. "Mutation-Driven Generation of Unit Tests and Oracles". In: *IEEE Transactions on Software Engineering* 38.2 (2012), pp. 278–292.

[31] José Campos et al. "An Empirical Evaluation of Evolutionary Algorithms for Unit Test Suite Generation". In: *Information and Software Technology* 104 (Aug. 2018). DOI: `10.1016/j.infsof.2018.08.010`.

[32] Davide Spadini, Maurí cio Aniche, and Alberto Bacchelli. "PyDriller: Python framework for mining software repositories". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. New York, New York, USA: ACM Press, 2018, pp. 908–911. ISBN: 9781450355735. DOI: `10.1145/3236024.3264598`. URL: `http://dl.acm.org/citation.cfm?doid=3236024.3264598`.