A Natural Language Programming Approach for Requirements-based Security Testing

Phu X. Mai, Fabrizio Pastore, Arda Goknil, Lionel C. Briand
SnT Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
{xuanphu.mai, fabrizio.pastore, arda.goknil, lionel.briand}@uni.lu

***Abstract*—To facilitate communication among stakeholders, software security requirements are typically written in natural language and capture both positive requirements (i.e., what the system is supposed to do to ensure security) and negative requirements (i.e., undesirable behavior undermining security).**

In this paper, we tackle the problem of automatically generating executable security test cases from security requirements in natural language (NL). More precisely, since existing approaches for the generation of test cases from NL requirements verify only positive requirements, we focus on the problem of generating test cases from negative requirements.

We propose, apply and assess Misuse Case Programming (MCP), an approach that automatically generates security test cases from misuse case specifications (i.e., use case specifications capturing the behavior of malicious users). MCP relies on natural language processing techniques to extract the concepts (e.g., inputs and activities) appearing in requirements specifications and generates executable test cases by matching the extracted concepts to the members of a provided test driver API. MCP has been evaluated in an industrial case study, which provides initial evidence of the feasibility and benefits of the approach.

***Index Terms*—System Security Testing, Natural Language Requirements, Natural Language Processing (NLP).**

I. Introduction

Software security has become a major concern, with the effect that a lot of attention is placed on the verification of the compliance of software systems with their security requirements [1]. Security requirements focus on both security properties of the system and potential security threats [2], [3], [4], [5], [6]. For instance, in use case-driven methodologies [4], [5], [7], security use cases describe the security properties of the system (e.g., user authentication) while misuse cases describe malicious activities (e.g., bypassing the authorization schema).

Security testing is driven by requirements [8] and, consequently, can be divided in two categories [9], [10]: (1) security functional testing validating whether the specified security properties are implemented correctly, and (2) security vulnerability testing addressing the identification of system vulnerabilities. Although several security testing approaches have been proposed [10], the automated generation of security test cases from security requirements remains limited in industrial settings. Security test cases are manually crafted by engineers who rely on automated tools for a limited set of activities (e.g., input generation to discover SQL injection vulnerabilities [11]).

Most security testing approaches focus on a particular vulnerability (e.g., buffer overflows [12], [13] and code injection vulnerabilities [14], [15]). These approaches deal with the generation of simple inputs (e.g., strings, files), and cannot be adopted to verify that the system is not prone to complex attack scenarios involving several interactions among parties, e.g., stealing an invitation e-mail to register multiple fake users on a platform. Model-based approaches are capable of generating test cases based on interaction protocol specifications [16], [17] and thus can potentially generate test cases for such complex attack scenarios [18]. They require formal models, which limits their adoption in industrial settings. Unfortunately, engineers tend to avoid such models because of the costs related to their development and maintenance, especially in contexts where system requirements in Natural Language (NL) are already available. There are approaches that generate functional system test cases from NL requirements [19], [20], [21], [22], [23]. However, these approaches can be adopted in the context of security functional testing, but not security vulnerability testing since they generate test cases only for the intended behavior of the system. In contrast, security vulnerability testing deals with the problem of simulating the behavior of a malicious user. Our goal in this paper is to enable automated security vulnerability test case generation from NL requirements. Our motivation is to have a systematic way to identify threats, to test whether they can be exploited, and to automate testing relying exclusively on artifacts that can be realistically expected in most environments.

In this paper, we propose, apply and assess Misuse Case Programming (MCP), an approach that generates security vulnerability test cases from misuse case specifications. To generate executable test cases from misuse case specifications we employ some concepts of *natural language programming*, a term which refers to approaches automatically generating software programs (e.g., executable test cases) from NL specifications [24], [25]. To enable the automated generation of executable test cases, MCP assumes that security requirements are elicited according to a misuse case template that includes keywords to support the extraction of control flow information. Our MCP prototype currently relies on the Restricted Misuse Case Modeling (RMCM) template [26], [4], which presents these characteristics. To interact with the system under test, MCP requires a test driver API that implements basic security testing activities (e.g., requesting a URL).

The natural language programming solution implemented by MCP includes an initial Natural Language Processing (NLP) step in which MCP derives models that capture the control flow of the activities described in a misuse case specification. MCP then translates the derived models into sequences of executable instructions (e.g., invocations of the test driver API's functions) that implement the malicious activities. To this end, we adapt the idea, developed by other works [27], [28], [29], [30], of combining string similarity and ontologies [31] to generate test cases from misuse case specifications. Similarly to other approaches, MCP builds an ontology that captures the structure of the given test driver API and generates executable instructions by looking for nodes in the ontology that are similar to phrases in NL requirements. The specificity of MCP is that it integrates additional analyses required to enable automated testing, which include the identification of test inputs, the generation of test input values and the generation of test oracles.

We successfully applied and evaluated our approach to an industrial case study in the medical domain, thus showing evidence that the approach is practical and beneficial to automatically generate test cases detecting vulnerabilities in industrial settings.

This paper is structured as follows. Section II provides the background techniques on which this paper builds the proposed test case generation approach. Section III discusses the related work. In Section IV, we present an overview of the approach. Sections V to VIII provide the details of the core technical parts of our approach. Section IX presents our industrial case study. We conclude the paper in Section X.

1 **MISUSE CASE** Bypass Authorization Schema
2 **Description** The MALICIOUS user accesses resources that are dedicated to a user with a different role.
3 **Precondition** For each role available on the system, the MALICIOUS user has a list of credential of users with that role, plus a list functions/resources that cannot be accessed with that role.
4 **Basic Threat Flow**
5 1. FOREACH role
6 2. The MALICIOUS user sends username and password to the system through the login page
7 3. FOREACH resource
8 4. The MALICIOUS user requests the resource from the system.
9 5. The system sends a response page to the MALICIOUS user.
10 6. The MALICIOUS user EXPLOITS the system using the response page and the role.
11 7. ENDFOR
12 8. ENDFOR
13 **Postcondition:** The MALICIOUS user has executed a function dedicated to another user with different role.
14 **Specific Alternative Threat Flow (SATF1)**
15 RFS 4.
16 1. IF the resource contains a role parameter in the URL THEN
17 2. The MALICIOUS user modifies the role values in the URL.
18 3. RESUME STEP 4.
19 4. ENDIF.
20 **Postcondition:** The MALICIOUS user has modified the URL.
21 **Specific Alternative Threat Flow (SATF2)**
22 RFS 4.
23 1. IF the resource contains a role parameter in HTTP post data THEN
24 2. The MALICIOUS user modifies the role values in the HTTP post data.
25 3. RESUME STEP 4.
26 4. ENDIF.
27 **Postcondition:** The MALICIOUS user has modified the HTTP post data.
28 **Specific Alternative Flow (SAF1)**
29 RFS 6
30 1. IF the response page contains an error message THEN
31 2. RESUME STEP 7.
32 3. ENDIF.
33 **Postcondition** The malicious user cannot access the resource dedicated to users with a different role.

Fig. 1. 'Bypass Authorization Schema' misuse case specification.

II. Background

In this section we present the background regarding the elicitation of security requirements (Section II-A), natural language processing (Section II-B) and natural language programming (Section II-C).

A. Eliciting Misuse Cases with RMCM

MCP assumes that misuse cases are elicited according to a format that includes keywords capturing control flow information. In our implementation we rely on the RMCM format [26], [4], which is briefly presented in this section.

Restricted Misuse Case Modeling (RMCM) is a use case-driven modeling method that supports the specification of security requirements in a structured and analyzable form [26], [4]. In RMCM, functional requirements are expressed as use cases, while security requirements are captured in security use cases and misuse cases (see an example misuse case in Fig. 1).

RMCM extends the Restricted Use Case Modeling method (RUCM) [32] originally proposed for use case specifications. RUCM is based on a template and restriction rules, reducing ambiguities and incompleteness in use cases. It has been successfully applied in many domains [33], [34], [35], [36], [37], [38], [39], [40]. This section does not distinguish between the original RUCM features and the extensions for RMCM, and we furthermore focus on misuse case specifications only.

RMCM provides means to characterize basic and alternative flows of activities in misuse case specifications as *Basic Threat Flow*, *Specific/Bounded/Global Alternative Flow* and *Specific/Bounded/Global Alternative Threat Flow* (see [26], [4] for the details). Threat flows specify unwanted incidents. A basic threat flow describes a nominal scenario for a malicious actor to harm the system (Lines 4 - 13 in Fig. 1). It contains misuse case steps and a postcondition.

Alternative flows describe alternative and valid execution scenarios; in other words, they capture failed attacks (e.g., Lines 28 - 33 in Fig. 1). Alternative *threat* flows describe alternative attack scenarios. For instance, in Lines 14 - 20 in Fig. 1, the specific alternative threat flow *SATF1* describes another successful attack scenario where the resource contains a role parameter. A specific alternative flow always depends on a specific step of the reference flow. A bounded alternative flow refers to more than one flow step while a global alternative flow refers to any step in the specification. For specific and bounded alternative flows, the `RFS` keyword is used to specify the reference flow steps (e.g., Line 31 in Fig. 1). The `IF .. THEN` keyword describes the conditions under which alternative (threat) flows are taken (e.g., Line 16 in Fig. 1).

B. Natural Language Processing (NLP)

NLP techniques extract structured information from documents in NL [41]. They implement a pipeline that executes multiple analyses, e.g., tokenization, morphology analysis, syntax analysis, and semantic analysis. Each pipeline step produces results based on the output of the previous step.

{The malicious user}$_{A0}$ {sends}$_{verb}$ {dictionary values}$_{A1}$ {to the system}$_{A2}$ {through the username and password fields}$_{AM-MNR}$

Fig. 2. Example SRL tags generated by CNP.

TABLE I
SOME OF THE PROPBANK ADDITIONAL SEMANTIC ROLES.

Identifier	Definition
AM-LOC	Indicates a location.
AM-MNR	Captures the manner in which an activity is performed.
AM-MOD	Indicates a modal verb.
AM-NEG	Indicates a negation, e.g. 'no'.
AM-PRD	Secondary predicate with additional information about A1.

In this paper, we rely on Semantic Role Labeling (SRL) [41]. SRL is a semantic analysis technique that determines the roles played by the phrases in a sentence, e.g., the actor affected by an activity. For the sentences "The system starts" and "The system starts the database", SRL determines that the actors affected by the actions are *the system* and *the database*, respectively. The component that is started coincides with the subject in the first sentence and with the object in the second sentence although the verb "to start" is used with an active voice in both. Therefore, this information cannot be captured by other NLP techniques like POS tagging and dependency parsing. The SRL roles can be effectively used to represent the meaning of a sentence in a structured form [42], which we need to generate API calls from a misuse case step.

To perform SRL, we rely on the CogComp NLP pipeline [43] (hereafter CNP), which has shown to be effective in our prior research [42]. CNP tags the words in a sentence with keywords (e.g., *A0*, *A1*, *A2*, *AN*) to indicate the roles according to the PropBank model [44]. *A0* indicates who (or what) performs an action, while *A1* indicates the actor most directly affected by the action. The other roles are verb-specific despite some commonalities (e.g., *A2* which is often used for the end state of an action). The PropBank model has also some other verb-independent roles (see Table I). They are labeled with general keywords and match adjunct information in different sentences, e.g., *AM-NEG* indicating a negation.

Fig. 2 shows the SRL output for an example misuse case step. The phrase "The malicious user" represents the actor who performs the activity (tagged with *A0*); the phrase "dictionary values" is the actor affected by the verb (i.e., tagged with *A1*). The phrase "to the system" is the final location (tagged with *A2*), while the last chunk of the sentence represents the manner (tagged with *AM-MNR*) in which the activity is performed.

C. Natural Language Programming

Natural language programming refers to a family of approaches that automatically generate programs from NL specifications (e.g., [27], [28], [29], [30], [45]). MCP has been inspired by techniques for building NL interpreters [27], [28], [29], [30], in particular NLCI, a technique accepting action commands in English to translate them into executable code.

NLCI [30] translates NL sentences describing the system activities into sequences of API calls that implement the intended actions. To this end, it requires an ontology that captures the structure of the given API. To identify the API calls, NLCI relies on a scoring mechanism using the string similarity between the terms in the sentence and the method and parameter names in the API.

Although inspired by NLCI, MCP differs from NCLI in many ways: MCP directly addresses security testing, MCP adopts a different scoring mechanism, and, finally, MCP supports the generation of assignment statements and (input) data structures (i.e., dictionaries), which is not supported by NLCI (see Section VIII).

III. RELATED WORK

Security testing verifies the compliance of a software system with its security requirements [10], [46], [47], which, in turn, can be given as *positive requirements*, specifying the expected functionality of a security mechanism, and as *negative requirements*, specifying what the system should not do [8], [9]. For instance, a positive security requirement is "a user account is disabled after five unsuccessful login attempts", while a negative security requirement is "a malicious customer should not be able to access resources that are dedicated to users with a different role (e.g., employee)". This classification of security requirements is reflected in security testing [8], [9], [10]: (1) security functional testing validating whether the specified security properties are implemented correctly, and (2) security vulnerability testing addressing the identification of system vulnerabilities. Security vulnerability testing mimics attackers who aim to compromise the security properties of the system (e.g., confidentiality, integrity, and availability) [10], [48]. Security vulnerability testing requires specific expertise for simulating attacks (e.g., identifying risks in the system and generating tests driven by those risks), which makes test case generation and execution difficult to automate [49].

Most vulnerability testing approaches focus on a particular vulnerability like buffer overflow and code injection vulnerabilities. For instance, Appelt et al. [15] present an automated approach that generates test inputs for SQLi attacks, while Tripp et al. [14] provide a learning algorithm for black-box detection of cross-site scripting (XSS) vulnerabilities. Ognawala et al. [13] present a tool that uses symbolic execution to detect memory out-of-bounds/buffer overflow vulnerabilities caused by unhandled memory operations in a program. Those approaches support engineers for a limited set of attack-related activities (e.g., input generation for an SQL injection vulnerability), and cannot be adopted to generate executable test cases for complex attack scenarios involving articulate interactions among parties, e.g., stealing an invitation e-mail to register multiple fake users on a platform. In contrast, MCP enables engineers to specify such attack scenarios in misuse case specifications and automatically generates executable test cases from the specifications.

Model-based testing approaches are capable of generating test cases based on interaction protocol specifications [16], [17] and thus can potentially generate test cases for complex attack scenarios [18]. Model-based security testing is a relatively new research field [46], where some approaches have been proposed for security vulnerability testing (e.g., [50], [51], [52], [53], [54], [55], [56], [57], [58], [59]). For instance, Marback et al. [53] propose a model-based security testing approach that automatically generates security test sequences from threat trees. Wimmel and Jürjens [60] present an approach that generates security test sequences for vulnerabilities from a formal model supported by the CASE tool AutoFocus. Whittle et al. [61] provide another approach that generates test sequences from attack scenarios in UML sequence and interaction overview diagrams. In these approaches, however, engineers have to transform the generated test sequences into executable tests manually, thus leading to limited benefits.

Xu et al. [62], [63] introduce the MISTA tool that automatically generates executable vulnerability test cases from formal threat models (i.e., Predicate/Transition - PrT nets). Jürjens [54] relies on some security extensions of UML, i.e., UMLsec [64], [65], [66], to generate security vulnerability test cases from detailed UML statecharts capturing control and data-flow. Bertolino et al. [50] present a model-based approach for the automatic generation of test cases for security policies specified in a process algebra language. All these approaches require detailed formal models, which limits their adoption in industrial settings. In fact, engineers tend to avoid such detailed models because of the costs related to their development and maintenance, especially in contexts where system requirements are already available in NL.

There are approaches that generate functional system test cases from NL requirements (e.g., [19], [20], [21], [22], [23]). For instance, Wang et al. [21], [22] automatically generate functional test cases from use case specifications written in RUCM. These approaches can be employed in the context of security functional testing, but not security vulnerability testing since they generate test cases only for the intended system behavior. Khamaiseh and Xu [67] present an approach that automatically builds, from misuse case specifications, PrT nets for the MISTA tool to automatically generate security vulnerability test cases. For the test code generation, test engineers have to provide some helper code and a model-implementation mapping description which maps the individual elements of a PrT net to their implementation constructs. In contrast, our approach does not need any helper code or model-implementation mapping.

IV. Overview of MCP

The process in Fig. 3 presents an overview of our approach. MCP takes as input a set of misuse case specifications and a test driver API implementing the functions required to test the system (e.g., functions that load URLs). The MCP tool prototype includes a generic test driver API for Web testing that can be extended for system specific operations. The input misuse case specifications should conform to a template which enforces (i) the use of simple sentences to facilitate NLP, (ii) the use of keywords to capture the control flow, and (iii) the use of attack keywords to specify which inputs should be generated according to predefined attack patterns. These are some of the characteristics of the RMCM template, though MCP may work with other templates.

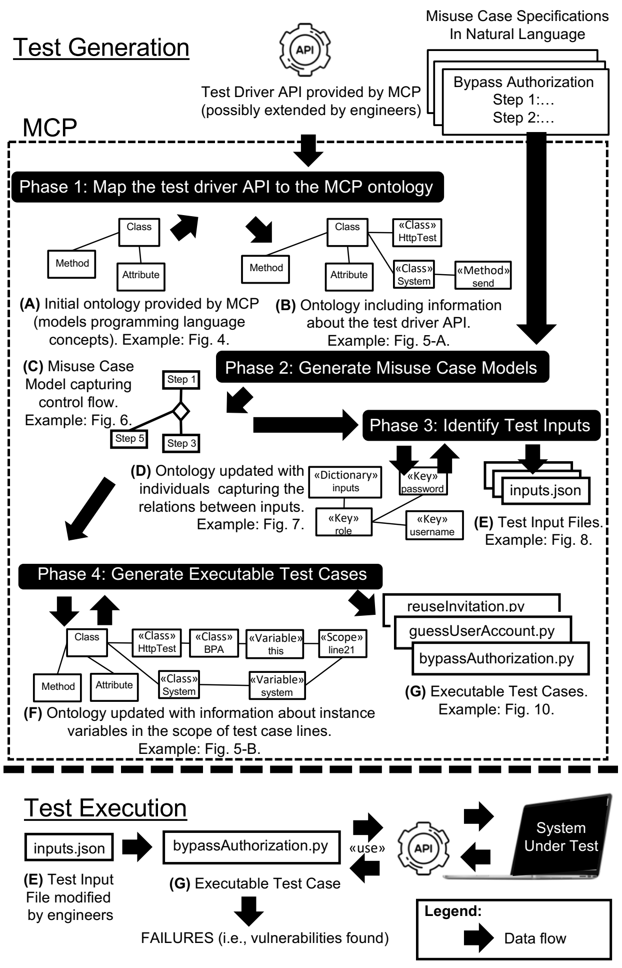
Test Generation
Misuse Case Specifications In Natural Language
API
Bypass Authorization Step 1:... Step 2:...
Test Driver API provided by MCP (possibly extended by engineers)
MCP
Phase 1: Map the test driver API to the MCP ontology
Class
Method
Attribute
«Class» HttpTest
«Class» System
«Method» send
(A) Initial ontology provided by MCP (models programming language concepts). Example: Fig. 4.
(B) Ontology including information about the test driver API. Example: Fig. 5-A.
(C) Misuse Case Model capturing control flow. Example: Fig. 6.
Step 1
Step 5
Step 3
Phase 2: Generate Misuse Case Models
Phase 3: Identify Test Inputs
(D) Ontology updated with individuals capturing the relations between inputs. Example: Fig. 7.
«Dictionary» inputs
«Key» password
«Key» role
«Key» username
inputs.json
(E) Test Input Files. Example: Fig. 8.
Phase 4: Generate Executable Test Cases
«Class» BPA
«Variable» this
«Scope» line21
«Variable» system
reuseInvitation.py
guessUserAccount.py
bypassAuthorization.py
(F) Ontology updated with information about instance variables in the scope of test case lines. Example: Fig. 5-B.
(G) Executable Test Cases. Example: Fig. 10.
Test Execution
inputs.json
(E) Test Input File modified by engineers
bypassAuthorization.py
(G) Executable Test Case
«use»
System Under Test
FAILURES (i.e., vulnerabilities found)
Legend:
Data flow

Fig. 3. Overview of the MCP approach.

MCP generates as output a set of executable security test cases that rely on the provided test driver API to perform the activities described in the misuse case specifications.

An essential component of the approach is an automatically populated ontology (hereafter *MCP ontology*). An ontology is a graph that captures the types, properties and relationships of a set of individuals (i.e., the basic blocks of an ontology). MCP uses the ontology to model programming language and test infrastructure concepts (Label A in Fig. 3), to capture the relationships and structure of the classes of the test driver API (Label B), to capture the relationships between inputs (Label D), and to represent the variables declared in the generated test case (Label F). We employ an OWL ontology [68] instead of UML diagrams because OWL provides simple means to query the modeled data. The MCP ontology is populated and managed using Apache Jena [69].

To generate test cases from misuse case specifications, MCP works in four phases. In the first phase, *Map the test driver API to the MCP ontology*, MCP processes the test driver API and augments the MCP ontology with individuals that model the classes and functions belonging to the given test driver

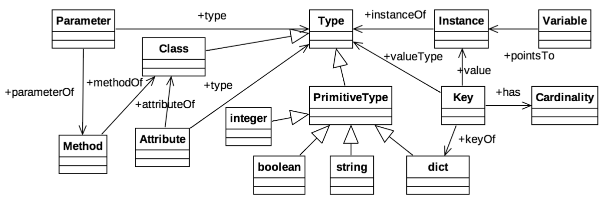

Fig. 4. Part of the MCP ontology capturing programming language concepts. We depict the ontology using a UML class diagram. UML classes model types of individuals. UML associations capture properties.

API (Labels A and B). In the second phase, *Generate misuse case models*, MCP relies on an NLP pipeline to derive models that explicitly capture the control flow implicitly described in misuse cases (Label C).

In the third phase, *Identify test inputs*, MCP determines the inputs to be sent to the system. It first identifies the input entities (Label D) and then prepares a configuration file that will be filled out by engineers with concrete input values to be used during testing (Label E). MCP can automatically generate the input values when these values can be derived by relying on predefined strategies (e.g., using grammars to generate inputs for code injection attacks [14], [15]).

In the fourth phase, *Generate executable test cases*, MCP automatically generates executable test cases from the misuse case models (Labels C and G). Each generated test case follows the control flow in the corresponding misuse case model and, for each step in the model, executes an operation implemented by the given test driver API. MCP employs a natural language programming solution to map each step in the misuse case model to an operation exposed by the test driver API. This solution maps NL commands (i.e., sentences in misuse case steps) to objects and methods of the provided API by retrieving information from the MCP ontology (Label F). While generating the test cases, the MCP ontology is augmented with individuals matching the variables declared in the test case.

We provide the MCP prototype with a set of predefined misuse case specifications derived from the OWASP testing guidelines [8]. These misuse cases can be reused (or adapted) across multiple projects; in addition, security analysts can write new, system specific misuse case specifications.

The rest of the paper provides a detailed description of each phase of MCP shown in Fig. 3, with a focus on how we achieved automation.

V. Mapping the Test Driver API to an Ontology

We provide an ontology (i.e., the MCP ontology) with concepts common to object-oriented programming languages (e.g., `Type`, `Class`, `Attribute`, `Method`, and `Parameter`). The MCP ontology also captures the concepts required to model the runtime behavior of a test case: `Instance` (i.e., an instance of a `Type`) and `Variable` (i.e., a program variable pointing to an instance). Fig. 4 shows part of the MCP ontology; we model programming language concepts as *types*, shown in Fig. 4 using UML classes.

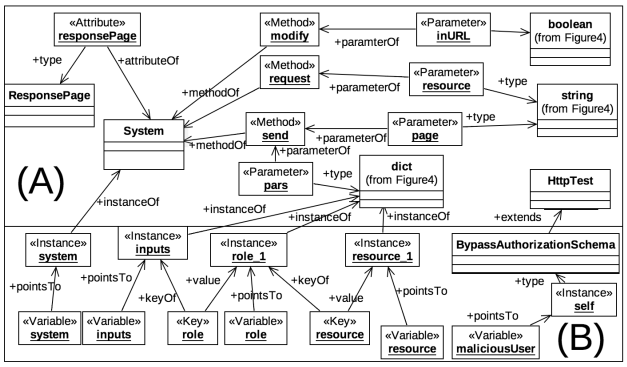

Fig. 5. Part of the MCP ontology populated when generating a test case for ‘Bypass Authorization Schema’. UML classes model types. UML objects model individuals. Part-A models the test driver API. Part-B models the variables in the scope of Line 21 of the test case in Fig. 10.

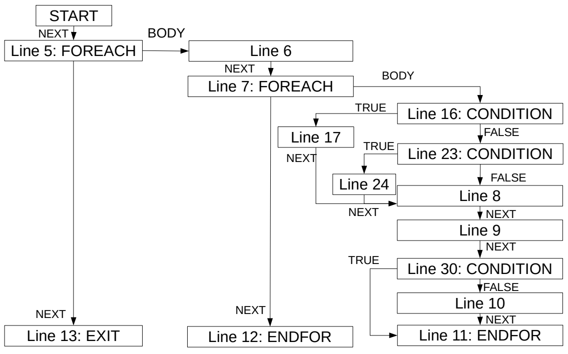

Fig. 6. Misuse case model for the misuse case specification in Fig. 1.

MCP automatically populates the MCP ontology with types and individuals that match the elements (e.g., methods) in the test driver API. For example, Fig. 5-A shows part of the populated MCP ontology that models the test driver API used in our case study. `System` is a type while `send` is an individual of type `Method` with the property `methodOf` set to `System` (we use association links to model properties).

Although our approach is language independent, the MCP prototype works with the test driver API in Python. Therefore, our prototype includes a Python component that relies on reflection to extract the names of classes, methods and method parameters from the given API. Although the Python programming language does not enforce the declaration of method parameter types, we assume that parameter types are captured by means of function annotations, a common practice adopted by Python programmers to document their software [70].

VI. Generating Misuse Case Models

MCP automatically generates a misuse case model for each misuse case specification (see Fig. 6). It employs an NLP pipeline that looks for control flow keywords (e.g., `IF .. THEN`, `FOREACH`, `RFS` and `RESUME` in RMCM) to build a model that captures the control flow implicitly specified in the misuse case specification. Each node in the model corresponds to a step in the misuse case specification. For simplicity, in

Fig. 6, we indicate only the type of the control flow nodes (i.e., CONDITION, FOREACH, ENDFOR and EXIT), while we report the line number of the step for the remaining nodes.

We do not provide details of the misuse case model generation because it is algorithmically simple and similar to the one adopted in our previous work [21]. Briefly, for each condition keyword encountered (i.e., IF .. THEN), MCP generates a control flow node that is linked to the first steps of the false and true branches. For each iteration keyword (i.e., FOREACH and DO .. UNTIL), it generates a node that is linked to the node corresponding to the step in the iteration body (see the arrows BODY in Fig. 6) and to the node matching the step executed after the iteration (see the arrows NEXT).

VII. Identifying Test Inputs

MCP determines input entities (e.g., 'role', 'password', 'username', and 'resource' in Fig. 1), input relationships (e.g., each 'username' is associated to a 'role'), and values to be assigned to input entities.

Consistent with RMCM, MCP assumes that input entities appear in misuse case steps with a verb that indicates that one or more entities are sent to the system under test by an actor (e.g., "The malicious user sends username and password to the system" and "The malicious user inserts the password into the system"). SRL is employed to automatically determine sentences indicating the sending of an entity to a destination. Depending on the verb, SRL usually tags destinations with A2 or AM-LOC (see Section II-B). Therefore, a sentence containing terms tagged with A2 or AM-LOC likely describes an input activity. In such sentences, MCP looks for the entities being sent, which match the terms tagged with A1 (i.e., the terms affected by the verb).

MCP automatically identifies relationships between input entities to avoid generating input values that may hamper the detection of vulnerabilities. For example, in Fig.1, we need to determine the roles associated to a username and password. This is necessary, for instance, to avoid using the username 'Mr. Phu', who is a patient, to simulate the behavior of a malicious doctor trying to access managers' data. If we use the username with the wrong role, the test case may not reveal that the system is vulnerable (e.g., malicious doctors might be able to access managers' data while patients might not).

MCP relies on the fact that a relationship between input entities can be derived from the control flow in a misuse case specification. For example, in Fig. 1, there is a one-to-many relationship between 'role' and 'resource' because multiple resources are requested with the same role (see Lines 5, 7 and 8). There is a one-to-one relationship between 'role' and 'username' because only one username is sent for each role (see Lines 5 - 6).

The MCP ontology is employed to capture input relationships by creating instances of the dict type. The dict type in the ontology is used to model the Python dictionary type, which maps keywords to values. Fig. 7 shows part of the populated MCP ontology that captures the input relationships in the misuse case in Fig. 1. The dict inputs individual

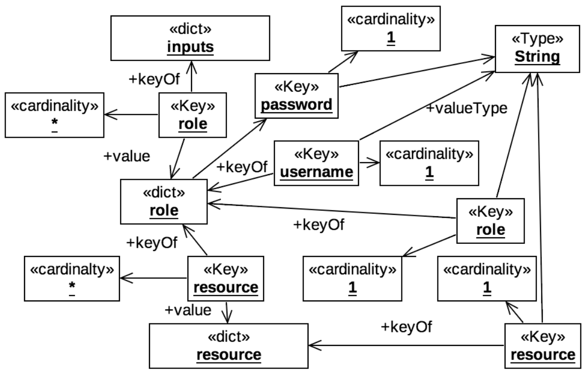

Fig. 7. Part of the populated MCP ontology for Fig. 1.

contains one Key individual for each input entity in the misuse case specifications (e.g., role, username and password). Also, it contains additional dict individuals for each entity appearing in an iteration (e.g., role) because these entities usually present containment relationships (e.g., each role has an associated username and password).

MCP relies on engineers to select input values. Automating the generation of input values is a challenge since it entails a complete understanding of system specifications. For example, to generate input values from the misuse case in Fig. 1, MCP needs existing users and roles, which cannot be automatically extracted without the architecture of the system under test. This information can be manually retrieved by engineers who know the system architecture and configuration.

To guide engineers in generating input values, MCP automatically generates a JSON file using the MCP ontology. The JSON format represents the content of dictionaries in textual form. The generated file contains input types (e.g., role) and placeholders to be replaced by engineers with values. Fig. 8 shows the JSON file generated from the MCP ontology in Fig. 7. Fig. 9 shows part of the same file with values. Note that engineers can specify more values for an input entity as suggested by the keyword ADD-MORE-ENTRIES; this is necessary to deal with iterations. The repeated entries might have a complex structure like in the case of role and resource which contain inner values (see Fig. 9).

To reveal some vulnerabilities, it is necessary to generate a large number of input values by using a predefined input generation strategy; this happens, for example, in the case of misuse cases that model attacks based on dictionary values or code injection (e.g., SQL injection). To assist engineers in such cases, MCP requires that an input generation strategy be indicated with a keyword in the misuse case specification (e.g., the keyword 'DICTIONARY VALUES' in Fig. 2).

To determine whether a predefined strategy needs to be used, MCP checks the terms tagged with A1 (i.e., the entity sent to the system) matching the keyword for the given strategy (e.g., 'DICTIONARY VALUES' in Fig. 2). If so, MCP looks for the terms tagged with AM-MNR (see Section II-B), which are the input entities to which dictionary values are assigned (e.g., 'username' and 'password' in the example above).

```
{"role": [
  {
    "password": "REPLACE-THIS-STRING",
    "role": "REPLACE-THIS-STRING",
    "username": "REPLACE-THIS-STRING"
    "resource": [
    {
      "resource": "REPLACE-THIS-STRING",
      "error_message": "REPLACE-THIS-STRING",
      "role_values": "REPLACE-THIS-STRING",
      "the_resource_contains_the_role_parameter_in_the_URL": "PUT-EXPRESSION",
      "the_resource_contains_the_role_parameter_in_the_HTTP_post_data": "PUT...
    },
     ADD-MORE-ENTRIES
    ],
  },
  ADD-MORE-ENTRIES
  ]
}
```

Fig. 8. Input file generated by MCP.

```
{"role": [
    {
      "role": "Doctor",
      "username": "phu@mymail.lu"
      "password": "testPassword1",
      "resource": [
       {
         "resource": "http://www.icare247.eu/?q=micare_invite&accountID=11"
         "error_message": "error",
    . . .
       },
      {
        "resource": "http://www.icare247.eu/?q=micare_skype/config&clientID=36"
        "error_message": "error",
        "the_resource_contains_the_role_parameter_in_the_URL": False,
        "the_resource_contains_the_role_parameter_in_the_HTTP_post_data": False
      }, ], }
    {
      "role": "Patient",
    . . .
```

Fig. 9. Part of the JSON file in Fig. 8 with input values.

VIII. Generating Executable Test Cases

MCP generates an executable test case for each misuse case specification. In the MCP prototype, each generated test case corresponds to a Python class that implements a method named `run`. Fig. 10 shows part of the test case generated for the misuse case in Fig. 1.

MCP declares and initializes three variables, `system`, `maliciousUser` and `inputs` (Lines 3, 4 and 5 in Fig. 10). The variable `system` refers to an instance of the class `System`, which provides methods that trigger the functions of the system under test (e.g., `request`). The variable `maliciousUser` refers to the test class, since the test class simulates the behavior of the malicious user. The variable `inputs` refers to a dictionary populated with the input values specified in the JSON input file. These three assignments are given in the MCP ontology with the individuals `maliciousUser`, `system` and `inputs` (see Fig. 5-B).

```
1   class bypassAuthorizationSchema(HTTPTester):
2     def run(self):
3       system = System(path=self.rootPath)
4       maliciousUser = self
5       inputs = self.loadInput("inputs.json")
6       roleIter = inputs["role"].__iter__()
7       while True:
8         try:
9           role = roleIter.__next__()
10          parameters = dict()
11          parameters["password"] = role["password"]
12          parameters["username"] = role["username"]
13          system.send("login page",parameters)
14          resourceIter = role["resource"].__iter__()
15          while True:
16            try:
17              resource = resourceIter.__next__()
18              if not eval(resource["the_resource_contains_a_role_
19                  parameter_in_the_URL"]):
20                if not eval(resource["the_resource_contains_a_role_parameter..
21                  system.request(resource)
22                  maliciousUser.responsePage = system.responsePage
23                  if not responsePage.contains( resource["error message"] )
24                    parameters = dict()
25                    parameters["resource"] = resource["resource"]
26                    parameters["role"] = role["role"]
27                    system.exploit(parameters)
28                  else:
29                    maliciousUser.abort("The MALICIOUS user CANNOT ex...")
```

Fig. 10. Part of the test case generated from the misuse case in Fig. 1.

MCP identifies the program elements (e.g., an API method) to be used in the generated test case based on string similarity. To do so, we employ a string similarity solution successfully used in our prior work [42], i.e., a function based on the Needleman-Wunsch string alignment algorithm [71]. To generate an executable test case, MCP processes all the nodes in the generated misuse case model (see Fig. 6). For each control flow node, MCP generates a control operation in the test case. For each other node, MCP generates both a method call and an assignment instruction by using the string similarity, and then selects one of them according to a scoring procedure.

In the following, we present the string similarity solution adopted by MCP, and the generation of method calls, assignments, control flow instructions and oracles.

A. String Similarity Measures

The Needleman-Wunsch string alignment algorithm maximizes the matching between characters by allowing for some degree of misalignment between them. The similarity degree adopted by MCP is computed as the percentage of matching characters in the aligned strings. In the rest of the paper, we write that a string s_a belonging to a set of strings S best matches a string s_t if the following holds:

$$\forall s : s \in S, D(s_a, s_t) \geq D(s, s_t) \ and \ D(s_a, s_t) \geq T$$

with D being the function for computing the degree of similarity of two strings and T being a threshold, set to 0.4 in our experiments, below which matching strings are excluded.

B. Generation of Method Calls

For each misuse case step, MCP aims to generate a method call that performs the activity described in the sentence. To achieve this goal, MCP must select the correct method to be invoked (i.e., a method with a proper name and parameters that belongs to a specific class instance) and identify which instance variables should be passed as argument.

To identify the class instance that should expose the method to be invoked, MCP queries the MCP ontology looking for individuals that best match, using similarity scores, the actors typically involved in a misuse case sentence: the system, the actor that performs the activity (SRL label `A0`), the actor affected by the activity (SRL label `A1`) or the actor that receives the inputs mentioned in the sentence (SRL labels `A2` and `AM-LOC`). For each selected individual, MCP looks for a method that is most likely to perform the activity described in the misuse case sentence.

MCP selects the method that maximizes a score that results from the average of: (S1) the string similarity degree between the method name and the verb in the sentence (to measure how well the method name matches the activity described by the sentence); (S2) the average string similarity degree of all the parameters with the best matching input entity (to determine if the method is supposed to work with the input entities appearing in the sentence); (S3) the percentage of terms (i.e., verb and noun phrases appearing in the misuse case sentence) that match the method parameters (to measure the completeness of the generated instruction, i.e., to what extent the concepts appearing in the sentence are reflected in the method call). The last point distinguishes MCP from other natural language programming solutions (e.g., [30]) since these do not measure the completeness of the generated instruction. MCP may also select a method call that best-matches the full sentence; this is done to properly deal with sentences describing specific attacks (e.g., "execute a network sniffing tool" which is implemented by the method `executeNetworkSniffingTool`, whose name includes the verb and the object in the sentence).

After identifying a method as the best candidate for a misuse case sentence, MCP generates the corresponding executable instruction as follows. First, MCP generates the program code corresponding to the invocation of the selected method (e.g., `system.request` in Line 21 in Fig. 10). Then MCP identifies the instance variables to pass as arguments; to this end, MCP queries the ontology looking for instance variables with the same type as the method parameter and with the best matching name. For example, in the case of Line 21 in Fig. 10, MCP selects the instance variable `resource`, which exactly matches the name of the parameter of the method `request`. If there is no variable matching the method parameter, MCP derives the value to be used directly from the text of the input entity that best matches the parameter name. This is necessary because the misuse case specification may include some of the values to be used during testing. If the parameter is a string, MCP sets the value to the name of the input entity (e.g., `"login page"` in Line 13). If the parameter is a boolean, MCP sets its value to `True` (this helps dealing with methods presenting boolean flags, e.g., the method `modify` shown in Fig. 5). Otherwise, MCP signals the missing parameter using a dedicated keyword in the generated code.

MCP also deals with API methods that declare an arbitrary number of parameters. This is the case of method `send` of the class `System` (Fig. 5-B), which is used to send a set of input values to the system under test and enables the specification of inputs to be sent according to the input page. For example, a login page may require two inputs (e.g., `username` and `password`), while the page for registering a new user may require more inputs. In Python, an arbitrary number of named arguments can be passed to a method by using a dictionary parameter. For this reason, in the presence of a dictionary parameter whose name does not match any input entity, MCP assumes that the dictionary parameter can be used to pass named arguments to the method. More precisely, MCP uses the identified dictionary parameter to pass input entities that do not match any other method parameter. These entities are taken into account when computing the score of the method (point S3 above). This is what occurs when MCP processes Line 6 of the misuse case specification in Fig. 1, which leads to generating the code appearing in Lines 10 - 13 in Fig. 10. The parameter `pars` of the method `system.send` is used by MCP to pass additional parameters to the method (i.e., `username` and `password`).

To simplify testing further, in the presence of test driver API methods requiring specific configuration parameters (e.g., the method `System.send` requires a mapping between a page name and its URL), engineers, instead of manually crafting a configuration file, can provide API methods that are invoked by MCP to automatically generate a file with the required configuration parameters.

C. Generation of Assignments

Assignment instructions are expected to be generated when some data (input or output) is exchanged between an actor and the system under test. MCP relies on SRL to identify the actor who performs the action (i.e., `A0` which is supposed to be the source of the data) and the final destination, which is captured by the SRL labels `A2` or `AM-LOC`. The data being transferred correspond to the terms tagged with `A1`. The assignment is then generated by looking for two instance variables that best match the terms tagged with `A0` (i.e., the data source for the right-hand side) and `A2` or `AM-LOC` (i.e., the destination for the left-hand side). The term tagged with `A1` (i.e., the data being moved) should then match an attribute of the objects referred by the selected variables. For example, the misuse case step "The system sends a response page to the malicious user" (Line 9 in Fig. 1) leads to the generation of the assignment in Line 22 in Fig. 10.

The score of the generated assignments is calculated by computing the mean of (1) the average string similarity degree for the terms used to identify the left-hand side and right-hand side of an assignment (to measure the likelihood that the selected terms match the concepts in the sentence) and (2) a value indicating the proportion of terms of the misuse case step that appear in the generated assignments (to measure the completeness of the generated assignments with respect to the concepts appearing in the step).

D. Generation of Control Flow Instructions

The generation of control flow instructions is straightforward and follows typical practices adopted in DSL-based and model-based code generation [72]. In this section, we focus on the generation of instructions from iterations and conditional sentences in NL, which is not addressed by DSL-based and model-based approaches.

Since iterations (i.e., sentences containing the keyword `FOREACH`) are used to indicate that a sequence of activities is repeated for a given set of input entities, MCP generates a block of code that iterates over all the values of the input variable matching the input entity mentioned in the `FOREACH` sentence. For example, Lines 6 - 9 in Fig. 10 show that the test case iterates over the elements of the list named `role`.

Condition sentences, instead, are used to indicate that certain activities are performed when a given condition, written in NL, holds. In general, a condition in the test code can be used to evaluate the truth value of either runtime data (e.g., the value returned by a method call) or input data (e.g., a configuration parameter). To deal with the first case, MCP generates a method call that best matches the condition in NL (Line 23 in Fig. 10). If the condition sentence does not match any method call, MCP assumes that the condition works with test input parameters, and thus generates a condition instruction evaluating the truth value of an input entity that matches the sentence (Line 18 in Fig. 10). The name of the input entity is added to the JSON input file (see the parameters starting with `the_resource_contains` in Fig. 8).

E. Generation of Oracles

In executable test cases, an automated oracle is typically implemented by means of instructions that report a failure when a certain condition does not hold; this is, for example, what JUnit assertions do [73]. *MCP automatically generates oracles*; this is implicitly achieved during the generation of the executable test case because MCP generates code that matches all the use case steps, including conditions that check erroneous outputs (e.g., Line 30 in Fig. 1) and instructions indicating that the malicious user can exploit a vulnerability (e.g., Line 10 in Fig. 1).

For example, the condition instruction in Line 23 of the test case in Fig. 10 corresponds to Line 30 in Fig. 1 and determines whether the system was not able to detect an unauthorized access. The instruction in Line 27 of Fig. 10, which corresponds to Line 10 in Fig. 1, is used to report a failure. In the MCP prototype, the method `System.exploit`, which matches misuse case steps indicating that a malicious user exploits a vulnerability, is used to report a failure.

IX. Empirical Evaluation

We have performed an empirical evaluation to respond to the following research questions:

- RQ1. Does MCP correctly identify input entities?
- RQ2. Can MCP generate executable test cases from misuse case specifications?
- RQ3. How do the generated test cases compare to manual test cases in terms of effectiveness, soundness and costs?

Case Study System and Empirical Setup

We applied MCP to generate test cases for a healthcare software system developed in the context of the EU project EDLAH2 [74]. The EDLAH2 project is developing a set of gamification-based services to engage clients (elderly people) in physical, mental, and social activities.

The EDLAH2 system is a representative example of a modern user-oriented system leveraging the capabilities of mobile and wearable devices. The software components of the EDLAH2 system are developed by professional software engineers, with three to twenty years of experience.

The EDLAH2 engineers follow the RMCM methodology to capture security requirements because RMCM specifications are written in NL and thus ease communication among all the stakeholders. The EDLAH2 misuse case specifications include a total of 68 misuse cases which describe both general attack patterns derived from the OWASP guidelines [8], [75] and system specific attacks that leverage some characteristics of the EDLAH2 system. For example, one of the EDLAH2 misuse cases models a malicious user who generates multiple user accounts by stealing the token of the page for inviting new users. Over the past months, the misuse case specifications of the EDLAH2 system have been used to manually derive test cases (scripts for manual testing and executable test cases).

To evaluate MCP, we have developed a Java prototype, which is available for download along with the experimental data presented in this section [76]. We have used the MCP prototype to generate executable test cases from 12 misuse case specifications. We have selected 12 misuse cases targeting the Web interface and with the highest risk according to the OWASP risk rating methodology [8]. Nine of the test cases manually derived from the selected misuse cases enabled the identification of vulnerabilities in the past months.

To perform the experiments, we have used a test driver API that was developed to support the manual implementation of the test cases from the EDLAH2 misuse case specifications. The API consists of ten classes and 84 methods in total.

RQ1

MCP reports the input entities in the `JSON` input file, which we inspected to evaluate the capability of MCP to determine correct input entities. We measure precision and recall according to standard formula [77]. In our context, true positives coincide with input entities, identified by MCP, which are correct (i.e., necessary to perform the test). False positives are input entities that do not correspond to software inputs. False negatives are input entities required to perform the attack (e.g., an input that should be provided to a form field of a Web page), which have not been identified by MCP.

In total, MCP leads to 29 true positives (i.e, input entities correctly identified), one false positive, and three false negatives. The false positive is due to the fact that one input entity belongs to an activity that is executed under conditions that do not hold for the EDLAH2 system (this is the case of the input entity 'role values' which is used in 'Bypass Authorization Schema' only for systems with URLs including role parameters). The three false negatives are caused by a concept (i.e., `invitation request`) which corresponds to three distinct input entities for the system under test (i.e., `email`, `username` and `message`). Overcoming false positives and negatives has shown to be simple since we did not modify the generated test code, but simply removed and added entries from and to the JSON input file. *Precision and recall are particularly high*, 0.97 and 0.91 respectively, which will favor the adoption of the technique in industrial settings.

RQ2

We inspected the source code of the generated test cases to spot the presence of errors affecting the control instructions, assignments, method calls and parameters. We also counted the number of test cases successfully executed without runtime errors due to programming mistakes. To execute the test cases, we have filled out the MCP input files with the help of EDLAH2 engineers.

The test cases generated by MCP do not contain any programming error and, furthermore, were all successfully executed against the EDLAH2 system. The generated test cases, one for each misuse case, are not trivial, and include a total of 791 lines of code (101 max for a single test, 55 min), 172 method calls (23 max, 10 min), 44 assignments (5 max, 3 min), 260 method arguments (42 max, 12 min). A subset of 128 method invocations concern the test drive API methods, while the rest corresponds to general Python utility methods. The generated test cases have been delivered to our industrial partners and are used to test the EDLAH2 system.

RQ3

We compared the test cases automatically generated by MCP with the test cases manually derived by EDLAH2 engineers for the same set of misuse case specifications, and with respect to effectiveness, soundness, and costs.

A security test case is *effective* if it is capable of discovering vulnerabilities that affect the system and it is *sound* if it does not report false alarms. The test cases generated by MCP identified all the nine vulnerabilities detected with manual testing, which shows that MCP test cases are as effective as manual test cases. Note that all these vulnerabilities result from real errors committed by engineers during software development. The test cases generated by MCP did not lead to the identification of any false alarm, thus showing that the approach is sound.

We discuss *costs* by comparing the effort required to perform vulnerability testing using MCP with the effort required by manual testing. To manually implement executable test cases, engineers must read and understand the security specifications of the system, an activity that requires substantial effort. Also, the implemented test cases might be error-prone and difficult to maintain. In the case of MCP, engineers do not need to implement or maintain executable test cases, but they require a test driver API and security specifications in NL. Our previous research results have shown that experienced engineers find that writing security specifications according to a structured format is helpful to support communication among stakeholders [26], which motivates the adoption of the RMCM methodology. In the presence of RMCM specifications, the generation of vulnerability test cases can be fully automated by MCP. To give additional evidence of the benefits of MCP, we count the lines of code of nine test cases developed by EDLAH2 engineers based on nine misuse case specifications, which is 1523. Considering that EDLAH2 requirements include more than 60 misuse cases, the manual implementation of all the required test cases would become expensive because of the effort required to write hundreds of lines of code after carefully reading several requirements specifications. This further motivates the adoption of MCP.

A test driver is also required by the manually written test cases, including functional test cases. Since a project-specific test driver API is necessary for both functional and security testing, its development costs do not directly result from the adoption of MCP. In addition, we provide the MCP prototype with a general test driver API that can be used with different Web projects, thus further reducing API development costs.

In both MCP and manual testing, engineers need to identify the input values to be used during testing (e.g., URLs). In general, the number of input values required for MCP and manual test cases derived from the same set of misuse cases is similar since they both cover the same scenarios. For each of the 12 MCP test cases generated in our experiment, engineers provided, on average, 15 distinct input values (excluding dictionary values) in the JSON input files and 11 configuration parameters required by the test driver API methods.

Threats to validity. The main threat to the validity regards generalizability, since results are linked to the case study system considered and the selected misuse case specifications. To deal with this threat, we considered a complex case study system which is a representative of modern user-oriented services for both its architecture and the technologies adopted. Also, we considered misuse cases that enabled the detection of vulnerabilities caused by real mistakes, committed by experienced software engineers during software development, and thus representative.

X. Conclusion

In this paper, we presented MCP, an approach that automatically generates vulnerability test cases, that is test cases simulating attacks and aimed at uncovering security vulnerabilities. MCP focuses on contexts where security requirements are written in Natural Language (NL), which is a common case since NL facilitates communication among stakeholder as in our industrial case study.

MCP requires as input a set of misuse case specifications and a test driver API and automatically generates a set of executable test cases that simulate the activities described in the misuse case specifications. MCP is a natural language programming solution that automatically translates each step in the misuse case specifications into executable instructions. The identification of the instructions to execute relies on NLP techniques. These techniques enable the identification of concepts that match the elements of the test driver API to be used in the test cases. For example, the actor performing an activity usually corresponds to an instance of an API class that exposes a method matching the verb in the sentence. The matching between concepts in NL requirements and the API is enabled by string similarity and an ontology which is used to model the test driver API and the generated test case. MCP assumes a consistent use of terminology between misuse case specifications and test driver API, which is generally true for modern test-driven development approaches. Future work will include the handling of synonyms (e.g., [42]).

Empirical results with an industrial case study system in the healthcare domain include the automated identification of real vulnerabilities in the developed system, an indication of the effectiveness of MCP. Also, MCP reduces the effort required for performing security vulnerability testing since it automates the generation of executable test cases which are not trivial to manually implement.

Acknowledgment. This work has received funding from the National Research Fund, Luxembourg FNR/P10/03, INTER/AAL/15/11213850 and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277).

References

[1] J. Kauflin, "The Fast-Growing Job With A Huge Skills Gap: Cyber Security," https://www.forbes.com/sites/jeffkauflin/2017/03/16/the-fast-growing-job-with-a-huge-skills-gap-cyber-security/, 2017.

[2] C. Haley, R. Laney, J. Moffett, and B. Nuseibeh, "Security requirements engineering: A framework for representation and analysis," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 133–153, 2008.

[3] H. Mouratidis and P. Giorgini, "Secure tropos: a security-oriented extension of the tropos methodology," *International Journal of Software Engineering and Knowledge Engineering*, vol. 17, no. 2, pp. 285–309, 2007.

[4] X. P. Mai, A. Goknil, L. K. Shar, and L. C. Briand, "Modeling security and privacy requirements for mobile applications: a use case-driven approach," University of Luxembourg, Tech. Rep., 2017.

[5] A. L. Opdahl and G. Sindre, "Experimental comparison of attack trees and misuse cases for security threat identification," *Information and Software Technology*, vol. 51, pp. 916–932, 2009.

[6] B. Fabian, S. Gurses, M. Heisel, T. Santen, and H. Schmidt, "A comparison of security requirements engineering methods," *Requirements Engineering*, vol. 15, pp. 7–40, 2010.

[7] D. G. Firesmith, "Security use cases," *Journal of Object Technology*, vol. 2, no. 3, pp. 53–64, 2003.

[8] M. Meucci and A. Muller, "OWASP Testing Guide v4," https://www.owasp.org/images/1/19/OTGv4.pdf.

[9] G. Tian-yang, S. Yin-Sheng, and F. You-yuan, "Research on software security testing," *World Academy of Science, Engineering and Technology*, vol. 70, pp. 647–651, 2010.

[10] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner, "Chapter one - security testing: A survey," ser. Advances in Computers. Elsevier, 2016, vol. 101, pp. 1–51. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0065245815000649

[11] "SQLMap, testing tool focussing on the detection of SQL injections." http://sqlmap.org/.

[12] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *USENIX Security'13*, 2013, pp. 49–64. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534772

[13] S. Ognawala, M. Ochoa, A. Pretschner, and T. Limmer, "MACKE: Compositional analysis of low-level vulnerabilities with symbolic execution," in *ASE'16*, 2016, pp. 780–785. [Online]. Available: http://dx.doi.org/10.1145/2970276.2970281

[14] O. Tripp, O. Weisman, and L. Guy, "Finding your way in the testing jungle: A learning approach to web security testing," in *ISSTA'13*, 2013, pp. 347–357. [Online]. Available: http://doi.acm.org/10.1145/2483760.2483776

[15] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for sql injection vulnerabilities: An input mutation approach," in *ISSTA'14*, 2014, pp. 259–269. [Online]. Available: http://doi.acm.org/10.1145/2610384.2610403

[16] M. Veanes, C. Campbell, W. Schulte, and N. Tillmann, "Online testing with model programs," in *ESEC/FSE'13*, 2005, pp. 273–282. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081751

[17] J. L. Silva, J. C. Campos, and A. C. R. Paiva, "Model-based user interface testing with spec explorer and concurtasktrees," *Electronic Notes in Theoretical Computer Science*, vol. 208, pp. 77–93, Apr. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2008.03.108

[18] F. Lebeau, B. Legeard, F. Peureux, and A. Vernotte, "Model-based vulnerability testing for web applications," in *ICSTW'13*, 2013, pp. 445–452.

[19] A. L. L. de Figueiredo, W. L. Andrade, and P. D. L. Machado, "Generating interaction test cases for mobile phone systems from use case specifications," *SIGSOFT Software Engineering Notes*, vol. 31, no. 6, pp. 1–10.

[20] G. Carvalho, D. Falcão, F. Barros, A. Sampaio, A. Mota, L. Motta, and M. Blackburn, "NAT2TESTSCR: Test case generation from natural language requirements based on SCR specifications," *Science of Computer Programming*, vol. 95, pp. 275–297, 2014. [Online]. Available: http://dx.doi.org.proxy.bnl.lu/10.1016/j.scico.2014.06.007

[21] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal, "Automatic generation of system test cases from use case specifications," in *ISSTA'15*, 2015, pp. 385–396. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771812

[22] C. Wang, F. Pastore, A. Goknil, L. C. Briand, and M. Z. Z. Iqbal, "UMTG: a toolset to automatically generate system test cases from use case specifications," in *ESEC/FSE'15*, 2015, pp. 942–945.

[23] M. Kaplan, T. Klinger, A. M. Paradkar, A. Sinha, C. Williams, and C. Yilmaz, "Less is more: A minimalistic approach to UML model-based conformance test generation," in *ICST'08*, 2008, pp. 82–91.

[24] B. W. Ballard and A. W. Biermann, "Programming in natural language: "nlc" as a prototype," in *ACM'79*, 1979, pp. 228–237. [Online]. Available: http://doi.acm.org/10.1145/800177.810072

[25] O. Pulido-Prieto and U. Juárez-Martínez, "A survey of naturalistic programming technologies," *ACM Computing Surveys*, vol. 50, no. 5, pp. 70:1–70:35, Sep. 2017. [Online]. Available: http://doi.acm.org/10.1145/3109481

[26] P. X. Mai, A. Goknil, L. K. Shar, F. Pastore, L. C. Briand, and S. Shaame, "Modeling security and privacy requirements: a use case-driven approach," *Information and Software Technology*, 2018. [Online]. Available: https://doi.org/10.1016/j.infsof.2018.04.007

[27] C. Manning, M. Surdeanu, J. Bauer, J. abd Finkel, S. Bethard, and D. McClosky, "The stanford CoreNLP natural language processing toolkit," in *ACL'14*, 2014, pp. 55–60.

[28] V. Le, S. Gulwani, and Z. Su, "SmartSynth: Synthesizing smartphone automation scripts from natural language," in *MobiSys'13*, 2013, pp. 193–206. [Online]. Available: http://doi.acm.org/10.1145/2462456.2464443

[29] D. Guzzoni, C. Baur, and A. Cheyer, "Modeling human-agent interaction with active ontologies," in *Interaction Challenges for Intelligent Assistants, Papers from the 2007 AAAI Spring Symposium*. Stanford, California, USA: AAAI, 2007, pp. 52–59.

[30] M. Landhausser, S. Weigelt, and W. F. Tichy, "NLCI: a natural language command interpreter," *Automated Software Engineering*, vol. 24, no. 4, pp. 839–861, 2017. [Online]. Available: https://doi.org/10.1007/s10515-016-0202-1

[31] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowledge Acquisition*, vol. 5, no. 2, pp. 199 – 220, 1993. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1042814383710083

[32] T. Yue, L. C. Briand, and Y. Labiche, "Facilitating the transition from use case models to analysis models: Approach and experiments," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 1, pp. 1–38, 2013.

[33] J. Zhou, Y. Lu, K. Lundqvist, H. Lonn, D. Karlsson, and B. Liwang, "Towards feature-oriented requirements validation for automotive systems," in *RE'14*, 2014, pp. 428–436.

[34] T. Yue, S. Ali, and M. Zhang, "RTCM: A natural language based, automated, and practical test case generation framework," in *ISSTA'15*, 2015, pp. 397–408. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771799

[35] I. Hajri, A. Goknil, L. C. Briand, and T. Stephany, "Applying product line use case modeling in an industrial automotive embedded system: Lessons learned and a refined approach," in *MODELS'15*, 2015, pp. 338–347.

[36] ——, "Configuring use case models in product families," *Software and Systems Modeling*, vol. 17, no. 3, pp. 939–971, 2018.

[37] ——, "PUMConf: a tool to configure product specific use case and domain models in a product line," in *FSE'16*, 2016, pp. 1008–1012.

[38] ——, "Incremental reconfiguration of product specific use case models for evolving configuration decisions," in *REFSQ'17*, 2017, pp. 3–21.

[39] I. Hajri, A. Goknil, and L. C. Briand, "A change management approach in product lines for use case-driven development and testing," in *Poster and Tool Track at REFSQ'17*, 2017.

[40] I. Hajri, A. Goknil, L. C. Briand, and T. Stephany, "Change impact analysis for evolving configuration decisions in product line use case models," *Journal of Systems and Software*, vol. 139, pp. 211–237, 2018.

[41] D. Jurafsky and J. H. Martin, *Speech and Language Processing (3rd ed.)*, 3rd ed. Prentice Hall, 2017.

[42] C. Wang, F. Pastore, and L. Briand, "Automated generation of constraints from use case specifications to support system testing," in *ICST'18*, 2018.

[43] University of Illinois, "CogComp NLP Pipeline," 2017.

[44] M. Palmer, D. Gildea, and P. Kingsbury, "The proposition bank: An annotated corpus of semantic roles," *Computational Linguistics*, vol. 31, no. 1, pp. 71–106, 2005.

[45] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra, "Automating test automation," in *ICSE'12*, 2012, pp. 881–891.

[46] M. Felderer, P. Zech, R. Breu, M. Büchler, and A. Pretschner, "Model-based security testing: A taxonomy and systematic classification," *Software Testing, Verification and Reliability*, vol. 26, no. 2, pp. 119–148, 2016. [Online]. Available: http://dx.doi.org/10.1002/stvr.1580

[47] M. Hafner and R. Breu, *Security Engineering for Service-oriented Architectures*. Springer Science & Business Media, 2008.

[48] B. Arkin, S. Stender, and G. McGraw, "Software penetration testing," *IEEE Security & Privacy*, vol. 3, no. 1, pp. 84–87, 2005.

[49] B. Potter and G. McGraw, "Software security testing," *IEEE Security & Privacy*, vol. 2, no. 5, pp. 81–85, 2004.

[50] A. Bertolino, S. Daoudagh, F. Lonetti, E. Marchetti, F. Martinelli, and P. Mori, "Testing of PolPA authorization systems," in *AST'12*, 2012, pp. 8–14.

[51] A. Blome, M. Ochoa, K. Li, M. Peroli, and M. T. Dashti, "Vera: A flexible model-based vulnerability testing tool," in *ICST'13*, 2013, pp. 471–478.

[52] K. He, Z. Feng, and X. Li, "An attack scenario based approach for software security testing at design stage," in *ISCSCT'08*, 2008, pp. 782–787.

[53] A. Marback, H. Do, K. He, S. Kondamarri, and D. Xu, "A threat model-based approach to security testing," *Software: Practice and Experience*, vol. 43, no. 2, pp. 241–258, 2013.

[54] J. Jürjens, "Model-based security testing using UMLsec: A case study," *Electronic Notes in Theoretical Computer Science*, vol. 220, no. 1, pp. 93–104, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066108004453

[55] M. Masood, A. Ghafoor, and A. Mathur, "Conformance testing of temporal role-based access control systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 2, pp. 144–158, April 2010.

[56] D. Xu and K. E. Nygard, "Threat-driven modeling and verification of secure software using aspect-oriented petri nets," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 265–278, 2006.

[57] E. Martin and T. Xie, "A fault model and mutation testing of access control policies," in *WWW'07*, 2007, pp. 667–676. [Online]. Available: http://doi.acm.org/10.1145/1242572.1242663

[58] ——, "Automated test generation for access control policies via change-impact analysis," in *SESS'07*, 2007.

[59] E. Martin, T. Xie, and T. Yu, "Defining and measuring policy coverage in testing access control policies," in *ICICS'06*, 2006, pp. 139–158. [Online]. Available: https://doi.org/10.1007/11935308_11

[60] G. Wimmel and J. Jürjens, "Specification-based test generation for security-critical systems using mutations," in *ICFEM'02*, 2002, pp. 471–482.

[61] J. Whittle, D. Wijesekera, and M. Hartong, "Executable misuse cases for modeling security concerns," in *ICSE'08*, 2008, pp. 121–130.

[62] D. Xu, M. Tu, M. Sanford, L. Thomas, D. Woodraska, and W. Xu, "Automated security test generation with formal threat models," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 4, pp. 526–540, 2012.

[63] D. Xu, W. Xu, M. Kent, L. Thomas, and L. Wang, "An automated test generation technique for software quality assurance," *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 247–268, 2015.

[64] J. Jürjens, "UMLsec: Extending UML for secure systems development," in *UML'02*, 2002, pp. 412–425.

[65] ——, "Sound methods and effective tools for model-based security engineering with UML," in *ICSE'05*, 2005, pp. 322–331.

[66] ——, *Secure Systems Development with UML*. Springer Science & Business Media, 2005.

[67] S. Khamaiseh and D. Xu, "Software security testing via misuse case modeling," in *DASC/PiCom/DataCom/CyberSciTech'17*, 2017, pp. 534–541.

[68] "Web Ontology Language (OWL)," https://www.w3.org/OWL/.

[69] "Apache Jena Semantic Web and Linked Data toolset," https://jena.apache.org/.

[70] "Function annotations, https://www.python.org/dev/peps/pep-3107/."

[71] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, Mar. 1970.

[72] L. Bettini, *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*, 2nd ed. Packt Publishing, 2016.

[73] "JUnit Testing Framework," https://www.junit.org/.

[74] "EDLAH2: Active and Assisted Living Programme," http://www.aal-europe.eu/projects/edlah2/.

[75] "OWASP, Android Testing Guidelines." https://www.owasp.org/index.php/Android_Testing_Cheat_Sheet.

[76] "MCP, prototype tool and experimental data." https://sntsvv.github.io/MCP/.

[77] D. Lane, "Online statistics education: A multimedia course of study." [Online]. Available: http://onlinestatbook.com/