COSMOS

DevOps for Complex Cyber-physical Systems

Project Number 957254

D4.4 Prototype of a toolset for specification-based functional security testing of CPS

Version 1.0
23 December 2021
Final

Public Distribution

University of Luxembourg

Project Partners: **Aicas, Delft University of Technology, GMV Skysoft, Intelligentia, Q-media, Siemens, Siemens Healthcare, The Open Group, University of Luxembourg, University of Sannio, Unparallel Innovation, Zurich University of Applied Sciences**

Project Partner Contact Information

Aicas James Hunt Emmy-Noether-Strasse 9 76131 Karlsruhe Germany Tel: +49 721 663 968 0 E-mail: jjh@aicas.com	**Delft University of Technology** Annibale Panichella Van Mourik Broekmanweg 6 2628 XE Delft Netherlands Tel: +31 15 27 89306 E-mail: a.panichella@tudelft.nl
Intelligentia Davide De Pasquale Via Del Pomerio 7 82100 Benevento Italy Tel: +39 0824 1774728 E-mail: davide.depasquale@intelligentia.it	**GMV Skysoft** José Neves Alameda dos Oceanos Nº 115 1990-392 Lisbon Portugal Tel. +351 21 382 93 66 E-mail: jose.neves@gmv.com
Q-media Petr Novobilsky Pocernicka 272/96 108 00 Prague Czech Republic Tel: +420 296 411 480 E-mail: pno@qma.cz	**Siemens** Birthe Boehm Guenther-Scharowsky-Strasse 1 91058 Erlangen Germany Tel: +49 9131 70 E-mail: birthe.boehm@siemens.com
Siemens Healthineers David Malgiaritta Siemensstrasse 3 91301 Forchheim Germany Tel: +49 9191 180 E-mail: david.malgiaritta@siemens-healthineers.com	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of Sannio Massimiliano Di Penta Palazzo ex Poste, Via Traiano I-82100 Benevento Italy Tel: +39 0824 305536 E-mail: dipenta@unisannio.it	**University of Luxembourg** Domenico Bianculli 29 Avenue J. F. Kennedy L-1855 Luxembourg Luxembourg Tel: +352 46 66 44 5328 E-mail: domenico.bianculli@uni.lu
Unparallel Innovation Bruno Almeida Rua das Lendas Algarvias, Lote 123 8500-794 Portimão Portugal Tel: +351 282 485052 E-mail: bruno.almeida@unparallel.pt	**Zurich University of Applied Sciences** Sebastiano Panichella Gertrudstrasse 15 8401 Winterthur Switzerland Tel: +41 58 934 41 56 E-mail: panc@zhaw.ch

Document Control

Version	Status	Date
0.1	Document outline	15 September 2021
0.2	Background ready	11 October 2021
0.3	Extensions to MCP	19 November 2021
0.4	Extensions to MST	1 December 2021
0.5	Results based on Catalog of MRs	10 December 2021
0.6	First full draft	16 December 2021
0.7	Applied comments and added Section 4.3	21 December 2021
1.0	QA review	23 December 2021

Table of Contents

1 Introduction **1**

2 Background and Related Work **4**

2.1 Security Testing of CPS 4

2.2 Metamorphic Testing for Security 7

2.2.1 Concepts of Metamorphic Testing 8

2.2.2 Metamorphic Security Testing - MST 9

2.2.3 SMRL: A DSL for Metamorphic Relations 10

2.2.4 SMRL to Java transformation 13

2.2.5 Data Collection 14

2.2.6 Execution of Metamorphic Relations 15

2.2.7 Catalog of Metamorphic Relations 16

2.2.8 MST-related work 18

2.3 Misuse Case Programming - MCP 20

2.3.1 Elicitation of Misuse Cases 20

2.3.2 Automated Generation of Security Vulnerability Test Cases 21

2.3.3 Providing Input and Configuration Values 23

3 COSMOS Security Testing Toolset Architecture **26**

4 Metamorphic Security Testing: Extended MST Component **28**

4.1 Extended MST Features 28

4.2 Definition of an extended catalog of MRs 34

4.2.1 Subjects of the Study 34

4.2.2 Analysis Procedure 35

4.2.3 Results 37

4.3 MST for Automotive 44

5 Misuse Case Programming: Extended MCP toolset **48**

5.1 Restricted Misuse Case Modelling 48

5.2 Natural Language Processing 49

5.3 Identification of Test Inputs 51

5.4 Generation of Executable Test Cases 52

6 Conclusion **54**

A CWEs covered by the Extended Catalog of Security MRs **55**

Executive Summary

Security testing can be divided into two categories: (1) security functional testing, which validates whether the security requirements of the system are implemented correctly, and (2) security vulnerability testing addressing the identification of system vulnerabilities. The former is performed by ensuring that documented software misuses are infeasible, the latter by automatically identifying new software misuses that break the security properties of the software.

In this document, we report about solutions that enable automating both; more precisely we extended two existing approaches: Metamorphic Security Testing (MST) and Misuse Case Programming (MCP). MST is a technique that ensures that both documented and unknown (or partially documented) software misuses are infeasible. It relies on a set of manually written specifications, which are called metamorphic relations (MRs). During the reported period, COSMOS has extended the available catalog of MRs thus enabling the identification of 100 vulnerability types concerning mistakes in the application of security design principles, which is result that is clearly valuable for software developers. To enable the implementation of the MR catalog, we have also extended the functionalities of the MST toolset. MCP is a technique that enables the automated generation of test cases based on documented software misuses (misuse case specifications in natural language); in the report, we describe the extensions performed to address some limitations of its NLP analysis thus enabling the processing of a wider set of misuse case specifications. In COSMOS, MCP will be used for the discovery of security vulnerabilities that cannot be discovered by MST.

Future work concerns extending MST and MCP to interact with CPS-specific interfaces and report on an ongoing empirical evaluation. Finally, we will address challenges related to minimizing testing cost and automatically extending security vulnerability testing capabilities by automatically generating MRs through machine learning.

1 Introduction

This report concerns *COSMOS Task T4.3: Development of Solutions for Detecting Security Vulnerabilities in CPS* (M1-M33, Leader: UoL. Contributors: UoS,TUD, QMA, TOG); we report about the period M1-M12, which concerned the definition of specification-based techniques for automated security testing.

Security testing can be divided into two categories [143, 59]: (1) security functional testing, which validates whether the security requirements of the system are implemented correctly, and (2) security vulnerability testing addressing the identification of system vulnerabilities. Based on the survey of Felderer et al., we define security functional testing as testing security mechanisms to ensure that their functionality is properly implemented. Security vulnerability testing is instead a form of risk-based testing motivated by understanding and simulating what an attacker may do. Also, security functional testing can be performed with classical functional testing techniques, whereas security vulnerability testing concerns the simulation of attacks based on security expertise. By attack we mean a sequence of interactions between a malicious user and the system under test, with the objective to compromise the security properties of a system.

Software security testing can thus be performed by ensuring that documented software misuses are infeasible, which is the objective of security functional testing, and by automatically identifying new software misuses that break the security properties of the software, which is the objective of security vulnerability testing.

Automation is a key for a successful software testing process. We have identified three challenges for automated security testing. The first challenge concerns ensuring, automatically or semi-automatically, that documented software misuses are infeasible. The second challenge is about the need for automated solutions to identify undocumented (or partially documented) misuses that break the security properties of the software. The third challenge concerns minimizing the cost of automated security testing, which is due to the need for identifying a large set of inputs and expected outputs.

The first challenge derives from the need for ensuring that the system allows only valid interactions (i.e., implements its security requirements) and is free from previously detected, or foreseen, vulnerabilities. It entails identifying methods to model valid software uses and software misuses that enable the automated generation of security test cases. The derivation of test cases from security requirements [72, 115, 120, 58] or threat models might be a solution; however, the literature lacks threat model for CPS that can be used to automatically derive test cases. In the CPS context, the generation of test cases often remains manual [56].

The second challenge concerns the discovery of unknown vulnerabilities through the identification of specific inputs (i.e., misuses) for which the software does not adhere to its security requirements. State-of-the-practice solutions are vulnerability scanners (e.g., Acunetix, Aircrack, Burp Suite, Intruder, Nexpose, Nessus); they automate the detection of vulnerabilities through the execution of attack patterns (i.e., sequences of inputs that lead to specific outputs when the software is not secure). They are effective in testing systems running on widely adopted frameworks (e.g., Web frameworks) but ineffective in the case of CPS; indeed, CPS often provide features for which attack patterns are not known. In addition, most automated techniques for security testing often focus on a particular vulnerability (e.g., buffer overflows [73, 119] and code injection vulnerabilities [144, 23]). These approaches deal with the generation of simple inputs (e.g., strings, files). Therefore, they cannot be adopted to verify that the system is not prone to complex attack scenarios involving several interactions among parties. Model-based approaches can generate test cases based on interaction protocol specifications [150, 137] and potentially produce test cases for complex attack scenarios [93]; however, they require executable models, which are not always adopted in industrial settings (see COSMOS deliverable report D4.1).

The third challenge is about the efficient and automated selection of test inputs and oracles. Since security vulnerabilities often manifest with a specific set of inputs, it might be necessary to test the system with a large set of test inputs, which might be exacerbated in the case of CPS because of the diverse nature of their inputs (e.g., continuous signals). Also, in the absence of strategies to automatically verify the output of the system (e.g., by automatically determining the expected output values), exhaustive testing remains infeasible.

In the reported period, COSMOS focussed on addressing the first two challenges above by relying on misuse case programming (MCP) and metamorphic security testing (MST).

MCP is a state-of-the-art solution previously developed by the University of Luxembourg [99]. It automatically generates an executable test case that simulates an attack described in a misuse case specification [120, 60] written using a constrained natural language. To generate the executable test cases, it relies on natural language processing (NLP) techniques to match specifications' sentences to functions exposed by a provided test API. Because of its characteristics, it can be used to address our first challenge above. However, MCP still requires the manual specification of test inputs and oracles, mainly because misuse case specifications describe high-level concepts but testing automation requires specific inputs to be tested; for example, a misuse case specification may contain the statement "*the malicious user requests a resource she does not have access to*", without providing the user name and the URL of the resource to access.

MST is another state-of-the-art solution previously developed by University of Luxembourg [98]. It relies on a set of manually written specifications, which are called metamorphic relations (MRs), capturing the relations between the output for a valid input (called source input) and the output for a follow-up input. Source inputs are sequences of input values that are valid according to the interaction protocol of the software under test (e.g., inputs exercised during functional testing). Follow-up inputs are derived by altering the source input as an attacker would do. The MST tool focuses on Web systems (i.e., systems providing a Web interface to end-users); it relies on a Web crawler to automatically derive source inputs for the system under test. Within COSMOS, MST will be extended to deal with different types of systems (e.g., CPS components exposing REST APIs or working with CAN buses). The possibility to automatically derive a large number of source and follow-up inputs enables MST to effectively simulate an attacker who tries to spot vulnerabilities by extensively testing the software. Moreover, MRs verify that expected relations between multiple inputs and outputs hold, hence can be used as oracles. Therefore, by automatically generating follow-up inputs and oracles, MST addresses the second challenge above.

Since MST can systematically test the system as an attacker would do, it can discover not only new software misuses that break the security properties of the software (second challenge) but also documented software misuses (first challenge). Therefore, since MST does not require the manual specification of test inputs and oracles, in COSMOS, we prioritized the development of MST over MCP; in COSMOS, MCP will be used only to test for attacks not covered by MST. One of our main objectives in COSMOS is therefore to define a catalog of MRs that cover a large set of vulnerability types, including CPS-specific ones.

Within the reported period, the COSMOS consortium has extended the catalog of MRs provided in the original MST work, to cover for a large set of vulnerabilities. More precisely, we focused on the definition of MRs that can discover vulnerabilities due to mistakes in the application of security design principles [126, 127], which we find to be a scenario that may occur during the development of a wide range of software systems, including CPSs. We considered all the 223 vulnerability types belonging to the above-mentioned category published by the MITRE corporation [9] and we derived a catalog of MRs that can discover 100 of these 223 vulnerabilities. When defining our MRs, we focused on the vulnerability types that may affect Web systems because Web systems are often used to access and control CPS; future work concerns extending our catalog further to include CPS-specific vulnerabilities. To enable the implementation of such catalog of MRs, we have also extended the MST toolset to include all the required features. Concerning MCP, we have addressed some limitations of its NLP analysis thus enabling the processing of a wider set of misuse case specifications.

This document describes a toolset that consists of five tools; we plan to release the toolset under an open-source license according to the exploitation plans of the project. For the time being, the software prototypes are available to COSMOS reviewers at the following URLs[1]:

- MCP tool, `https://gitlab.uni.lu/cosmos/mcp`
- MST tool, `https://gitlab.uni.lu/cosmos/mst`

[1]Please contact the consortium representatives and/or `fabrizio.pastore@uni.lu` to request access (GitHub account handle required).

- MST extended catalog of MRs, `https://gitlab.uni.lu/cosmos/mst_metamorphic_relations`
- MST crawler, `https://gitlab.uni.lu/cosmos/mst_crawljax`
- MST SMRL Eclipse plugin, `https://gitlab.uni.lu/cosmos/mst_smrl_eclipse`

This document proceeds as follows. Section 2 provides background information on security testing for CPS software and on the techniques integrated into COSMOS (i.e., MCP and MST). Section 3 provides an overview of the COSMOS architecture. Section 4 describes the extensions provided to the MST toolset and discusses the applicability of MST to discover security vulnerabilities caused by bad design practices, based on our extended catalog of MRs. Section 5 describes our extensions to MCP. Section 6 concludes this deliverable.

2 Background and Related Work

In this Section, we provide an overview of the state-of-the-art of security testing for CPS, highlighting the limitations of existing approaches (Section 2.1). Also, we provide an overview of Metamorphic Security Testing (Section 2.2) and Misuse Case Programming (Section 2.3), two approaches that will be extended by COSMOS to overcome the above-mentioned limitations.

2.1 Security Testing of CPS

Cyber Physical Systems (CPSs) are [25] networked systems of cyber (computation and communication) and physical (sensors and actuators) components that interact in real-time in a feedback loop with the possible help of human intervention. This adaptability in control-sharing between systems and humans can be seen [38] both as a trade-off between the capacity to react in unforeseen circumstances, and as a loss of reliability and predictive performance, being unintentional or deliberate sources of risks. CPS is a raising trend with multidisciplinary nature [156], focused mostly on protecting power grids, with only a few works on UAV, autonomous cars, etc.

Sensing components [153] are mainly deployed at the perception and transmission layers, they collect and sense information e.g., sensors, aggregators like Online Analytical Processing (OLAP), or actuators. Controlling components are deployed at the application layer, they monitor and control signals e.g., Programmable Logic Controller (PLC), Distributed Control System (DCS), or Remote Terminal Unit (RTU).

Security solutions are usually [25] designed for classical Information Technology (IT) systems but not for CPS. Moreover, due to constrained factors, most of the research focuses on the performance, stability, robustness and efficiency of CPSs rather than security.

Software testing concerns verifying that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain. Security testing aims to verify that software meets its requirements on security properties [59].

Security properties are usually confidentiality, integrity, and availability [56, 156], with sometimes more aspects like authentification and privacy [153], or authorization and non-repudiation [59, 83].

- *Integrity* ensures that sensitive data has not been accidentally or maliciously modified, altered, or destroyed. Deception attacks on integrity, especially false data injection attacks on sensors, are the most common [156].
- *Availability* guarantees timely, reliable access to data and information services for authorized users. Availability is compromised, for example, if the system crashes or in case of a Denial of Service (DoS) attack.
- *Confidentiality* ensures that sensitive information is not disclose to unauthorized individuals, processes, or devices. Disclosure attacks on confidentiality happen if the attacker can infer sensitive information from the system; for example, by eavesdropping (internal) communication channels, or side-channels.
- *Authentication* establishes the validity of a transmission, message, or originator, or a means of verifying an individual's authorization to receive specific categories of information.
- *Authorization* provides access privileges granted to a user, program, or process.
- *Non-repudiation* is the assurance that none of the partners taking part in a transaction can later deny of having participated.

A test oracle is a mechanism for determining the verdict of the test, i.e., the observed behavior of the system under test conform to the intended behavior. Security testing suffers from the *oracle problem*: how to distinguishing correct from incorrect behavior? What mechanism can be used for determining whether a test case passed or failed? This problem is partially addressed for availability because the detection of system

crashes and denial of services (i.e., lack of response from the system) can be easily automated. Metamorphic testing [98] (more details in Subsection 2.2) has shown to be effective in determining a number o security vulnerabilities (e.g., bypass authorization and authentication schema). However, the literature lacks extensive studies about the applicability of metamorphic testing in the security context; for example, there are no studies about the type of security vulnerabilities that can be automatically detected by means of metamorphic testing, which is in part addressed in this report document (see Section 4.2).

When test inputs and oracles are derived manually, which is common practice, they are often derived from models of security threats. In CPS context, *security threat analyses* usually [153] focus more on the perception and transmission layers than the application layer or the controllers that we are investigating in the COSMOS project. Hence the focus on security threats is usually more on physical (spoofing, sabotage, tracking, etc.) and communication (e.g., eavesdropping, replay, unauthorized access, or DoS attacks) threats than computation threats like authorization problems, injection vulnerabilities, or side-channel (e.g., timing [86]) attacks.

Security threats are not limited to the cyber environment and can cause physical hazards like human injuries, asset damages, or environmental impacts [38] e.g., Stuxnet in 2010, the Steel Mill cyber-attack in 2014, or TRITON in 2017. An example of threat modeling based on the most common attacks is STRIDE (Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege) [83], which is based on a data flow diagram of the components and the identification of security threats to the considered security properties.

Unfortunately, vulnerabilities and thus attacks depend on the considered type of system; therefore, generic threat models may not support well the verification of security requirements. Furthermore, there is a lack of test automation solutions that rely on threat models and can be applied to CPS. Indeed, the design and threat analyses are usually based on manual inspection of the documentation [29], and is applied manually [56] to testing CPS vulnerabilities. In the context of COSMOS, for example, we have an avionics system for air traffic control which can be subject to spoofing attacks, an automotive system requiring more automated pre-release security vulnerability checks, a railway system with security requirements, and a sensing system, requiring dedicated security testing to find security vulnerabilities in earlier stage of development. Below, we provide examples of threat modelling work for the automotive context and testing solutions based on such threat modelling, which are manual and therefore expensive.

Threat analysis takes into consideration the components of the system under test; in the case of automotive systems, common components are (1) electronic control units (ECUs), which control the state of the automatic transmission of the vehicle engine and manage the sensors inside the vehicle (e.g., body control and power-train control modules) and (2) networks interconnecting ECUs such as controller area network (CAN), local interconnect network (LIN), or FlexRay [84]. Small- and medium-sized vehicules include approximately 50 ECUs, up to 70 in luxury cars and 80 in cutting-edge vehicules [84], which contain and implement a total of up to 100 million code lines [83] to control safety-critical functionalities. The presence of a large number of components makes the attack surface large. Most of the surveys on CPS security [51, 54, 84] focus on attacks leveraging information sharing and wireless communications; this is the case for attacks on Intelligent Traffic Systems (ITS) [51] connecting Vehicular ad-hoc networks (VANETs) and the traffic system. Another survey discusses also in-vehicle vulnerabilities (e.g., vulnerabilities of ECUs and software vulnerabilities) [54], which depend on the traditional Electrical/Electronic Architecture (EEA) of Intelligent Connected Vehicles (ICVs) being based on the CAN. With CAN, the bandwidth is shared between components, which has outstanding advantages in cost-efficiency and flexibility, but makes the communication system vulnerable to attacks.

To our knowledge, the only work reporting about security testing driven by threat models in the automotive context is that of Dürrwang et al., who performed a *penetration test* of the Airbag-ECU through a CAN bus [56]. More precisely, the authors generated attack trees during threat analysis, based on the STRIDE approach [83] and used them to generate test cases revealing security vulnerabilities causing an unintended airbag deployment, which could harm occupants inside the vehicle[2]. This kind of manual approach is tedious and re-

[2]The attack has been assigned the MITRE vulnerability ID CVE-2017-14937, `https://nvd.nist.gov/vuln/detail/CVE-2017-14937`

quires expertise but general automated approaches are lacking. In COSMOS, we aim to address this problem by extending Metamorphic Security Testing (MST, presented in Section 2.2.2) and Misuse Case Programming (MCP, presented in Section 2.3), two approaches that support automated testing based on specifications of security attacks.

Beyond manual testing based on threat modelling, existing automated security testing solutions concern model-based security testing, code-based testing, penetration testing, and fuzzing; however, they had been seldom applied to CPS. We overview these approaches below.

Model-based security testing is based on requirements and design models created during the analysis and design phases [59]. In model-based testing, selected algorithms generate test cases from a set of models of the software under test or its environment. The benefits of model-based testing include early and explicit specification and review of system behavior, better test case documentation, the ability to automatically generate useful (regression) tests and control test coverage, improved maintenance of test cases as well as shorter schedules and lower costs [128]. For instance, the authors of [33] proposed a semi- automatic security testing approach for web applications based on a secure model built from abstract messages representing common actions. The model is secure as long as it does not violate confidentiality or authenticity goals, which is verified by using a model-checker. Security vulnerabilities are represented by modifying the model according to a predefined set of transformations (called *semantic mutation operators*, which leads to mutated models. The mutated models are used to generate attack trace using the model checker. The generated trace indicates the actions to be performed to exploit[3] the vulnerability. A system is vulnerable if it reaches the state expected by the mutated model. The main limitation of model-based approaches is that they require a security model of the system, which is often unavailable; furthermore, model-based approaches can often be used to generate abstract test cases only, which then need to be made executable through test adaptation, or test transformation approaches [149] thus increasing development cost.

Code-based testing is based on the source code of the software under test. For instance, symbolic execution of the code can be used on paths of interest to gather constraints that are solved to identify the input that satisfy such constraints and, therefore, generate concrete test cases. Symbolic execution tools include EXE [35], which generates inputs able to crash a given code, KATCH [105], which automatically tests code patches, SAVIOR [49], which prioritizes branches with the most potentially vulnerable locations in hybrid testing, or JOACO [142], which generates inputs leading to injection vulnerabilities. These tools can analyze all the control flows of a program thus achieving a high coverage of the software under test; however, the price a time-consuming exhaustiveness usually does not scale with the size of the software under test. In addition, typical constructs of CPS (e.g., floating point computations) are not well supported by existing tools [62].

Penetration testing targets the whole system from an attacker's point of view; it can be performed in a black-box or white-box fashion. Black-box penetration testing is performed by leveraging only information about the program input and output interfaces, so tends to be more realistic from an attacker's perspective. In white-box penetration testing the tester is provided with source code, full specifications, and documentation, thus enabling the identification of a larger set of vulnerabilities [29]. There are only a few penetration testing testing tools dedicated to CPSs. For instance, the Metasploit framework[4] has a few car modules. They allow Metasploit to be connected to actual cars through the CAN bus, or to virtual ones like Duraki's VirtualCar[5], a wrapper written in C that listens to the virtual CAN device and is able to parse, analyze, and transmit signals. The Metasploit modules allow basic features like getting vehicle information (e.g., vehicle identification number from the target module, engine speed, coolant temperature, diagnostic trouble codes), probing data points in a CAN packet, flooding a CAN interface with supplied frames, scanning the CAN bus for diagnostic modules, keeping the vehicle in a diagnostic state, performing hard reset in the ECU reset service identifier, sampling the targeted

[3] We use the term exploit to indicate when an input for the software under test lead to the execution of the faulty code (i.e., the code that makes the system vulnerable), and such execution results into a violation of a security property of the system.

[4] `https://www.metasploit.com/`

[5] `https://github.com/duraki/virtualcar`

module to flood the temperature gauge thus exploiting a vulnerability of the 2006 Chevrolet Malibu, moving the needle of the accelerometer and speedometer of a Mazda 2 instrument cluster, or checking for and prepping the pyrotechnic devices (like airbags, or battery clamps) of the targeted vehicle. But for the moment this framework capabilities are limited. Another tool is CANToolz[6], which supports end-user only in monitoring CAN communication and finding UDS services[7]. In general, the capabilities of existing penetration testing tools dedicated to automotive are limited. Similar considerations might be made also for the UAV context, where *dronesploit*[8], the best known tool, supports only a limited set of features [63]. Other industry sectors relevant for COSMOS, e.g., railway, are not covered by dedicated penetration testing tools. An exception is robotics, which has a large security community[9] and mature tools such as Alurity[10].

Fuzzing is a software testing technique that has received a lot of attention recently [112, 32, 102]. It is based on the idea of feeding malformed inputs to a program until it crashes. Random fuzzing uses a stream of random inputs, while mutation-based fuzzing generates variants of existing data samples, and generation-based fuzzing is based on a model of the input data or the vulnerabilities. Advanced fuzzing techniques combine these techniques with others like symbolic execution [34, 36]. The main limitation of fuzzing approaches is that they can be used to discover crashes, failure assertions, or compare the output of two software versions but they cannot detect vulnerabilities that manifest in a complex manner (e.g., determine confidentiality problems); also, there is limited work on the application of fuzzers with cyber-physical systems (CPS) [50, 136], which, however, concerns simplified prototypes or units of few lines of code. In COSMOS, we aim to address a range of vulnerabilities that is larger than the ones discovered by fuzzers and validate it with CPS case study subjects.

For our study, we consider the vulnerabilities listed in the CVE database [114], which is hosted by the MITRE [113] organization and is the result of an ongoing program to identify, define, and catalog publicly disclosed cybersecurity vulnerabilities. The CVE database is commonly used in security and software engineering research work to select case study subjects [31].

2.2 Metamorphic Testing for Security

In this deliverable we propose an extension to MST, which stands for Metamorphic Security Testing, an approach to automate security testing for Web systems through metamorphic testing.

Metamorphic Testing (MT) is a testing technique that has shown, in some contexts, to be very effective to alleviate the oracle problem [44, 96]. The *oracle problem* [28, 138, 122] refers to situations where it is extremely difficult or impractical to determine the correct output for a given test input. Security testing suffers from the oracle problem; indeed, security test cases seldom rely on automated test oracles, most often because it is infeasible or impractical to specify them due to a large number of test inputs. For instance, a security test case for the bypass authorization schema vulnerability should verify, for every specific user role, whether it is possible to access resources that should be available only to a user who holds a different role [109]. This type of vulnerability can often be discovered by verifying access to various resources with different privileges and roles. However, defining oracles is complicated by the need for identifying the resources that can be accessed by a user with a specific role and deciding if test outputs, which are often complex and non-deterministic, are consistent with expectations about accessibility; when expected outputs need to be identified for a large set of test inputs (e.g., for various resources, roles, and privileges), the identification of oracles is infeasible.

Since, within the reported period, the extension of MST has been the main focus for COSMOS Task 4.4, to make the deliverable self-contained, in the following, after an overview of metamorphic testing concepts, we

[6] `https://github.com/CANToolz/CANToolz`

[7] Unified Diagnostic Services (UDS) is a diagnostic communication protocol. It is used in ECUs and is specified in the ISO 14229-1 [75]

[8] https://dronesploit.readthedocs.io/en/latest/index.html

[9] See for example https://cybersecurityrobotics.net/

[10] https://aliasrobotics.com/alurity.php

present a detailed description of MST. The following sections are *an extension of what appears in the original MST paper [98]* and will be part of a future journal submission.

2.2.1 Concepts of Metamorphic Testing

MT is based on the idea that it may be simpler to reason about relations between outputs of multiple test executions, called metamorphic relations (MRs), than it is to specify its input-output behavior [132]. In MT, system properties are captured as MRs that are used to automatically transform an initial set of test inputs into follow-up test inputs. If the outputs of the system under test for the initial and follow-up test inputs violate the MR, it is concluded that the system is faulty.

The core of MT is a set of MRs, which are necessary properties of the program under test in relation to multiple inputs and their expected outputs [47]. MRs *resemble the traditional concept of program invariants, which are properties that hold at certain points in programs. However, the key difference is that an invariant has to hold for every possible program execution, whereas a metamorphic relation captures a property of inputs and outputs belonging to different executions* [132].

In MT, a single test case run requires multiple executions of the system under test with distinct inputs. The test outcome (pass or fail) results from the verification of the outputs of different executions against the MR.

As an example, let us consider an algorithm f that computes the shortest path for an undirected graph G. For any two nodes a and b in the graph G, it may not be practically feasible to generate all possible paths from a to b, and then check whether the output path is really the shortest path. However, a property of the shortest path algorithm is that the length of the shortest path will remain unchanged if the nodes a and b are swapped. Using this property, we can derive an MR, i.e., $|f(G, a, b)| = |f(G, b, a)|$, in which we need two executions of the function under test, one with (G, a, b) and another one with (G, b, a). The results of the two executions are verified against the relation. If there is a violation of the relation, then f is faulty.

We provide below basic definitions underpinning MT.

Definition 1 (Metamorphic Relation - MR). Let f be a function under test. A function f typically processes a set of arguments; we use the term *input* to refer to the set of arguments processed by the function under test. In our example, one possible input is (G, a, b). The function f produces an output. An MR is a condition that holds for any set of inputs $\langle x_1, ..., x_n \rangle$ where $n \geq 2$, and their corresponding outputs $\langle f(x_1), ..., f(x_n) \rangle$. MRs are typically expressed as implications.

In our example, the property of the target algorithm f is "the length of the shortest path will remain the same if the start and end nodes are swapped". The MR of this property is $(x_1 = (G, a, b)) \wedge (x_2 = (G, b, a)) \rightarrow |f(x_1)| = |f(x_2)|$.

Definition 2 (Source Input and Follow-up Input). An MR implicitly defines how to generate a *follow-up input* from a *source input*. A source input is an input in the domain of f. A follow-up input is a different input that satisfies the properties expressed by the MR. In our example, (G, a, b) and (G, b, a) are the source and follow-up inputs, respectively.

Follow-up inputs can be defined by applying *transformation functions* to the source inputs. The use of *transformation functions* in MRs simplifies the identification of follow-up inputs. In our example, a transformation function that swaps the last two arguments of the source input can be used to define the follow-up input:
$x_1 = (G, a, b) \wedge x_2 = swapLastArguments(x_1) \rightarrow |f(x_1)| = |f(x_2)|$
where $swapLastArguments$ is the transformation function.

Definition 3 (Metamorphic Testing - MT). MT consists of the following five steps:

1 Generate one source input (or more if required). In our example, a (random) graph G is generated; two vertices a and b in G are randomly selected for the source input.

2 Derive follow-up inputs based on the MR. In our example, the function $swapLastArguments$ is applied to (G, a, b).
3 Execute the function under test with the source and follow-up inputs to obtain their respective outputs. In our example, the shortest path function is executed two times with (G, a, b) and (G, b, a).
4 Check whether the results violate the MR. If the MR is violated, then the function under test is faulty.
5 Restart from (1), up to a predefined number of iterations.

2.2.2 Metamorphic Security Testing - MST

The process in Fig. 1 presents an overview of MST, the technique proposed by the University of Luxembourg to perform security testing by relying on a metamorphic testing approach[11] [98]. In Step 1, the engineer selects, from a catalog of predefined MRs, the relations for the system under test. The MST catalog of MRs was derived from the testing guidelines [109] edited by OWASP [17]. In addition, the engineer can also specify new relations by using the Security Metamorphic Relation Language (i.e., MST's DSL described in Section 2.2.3). Step 1 is manual. Once the engineer selects and/or specifies MRs, in Step 2, MST automatically transforms them into executable Java code (Section 2.2.4).

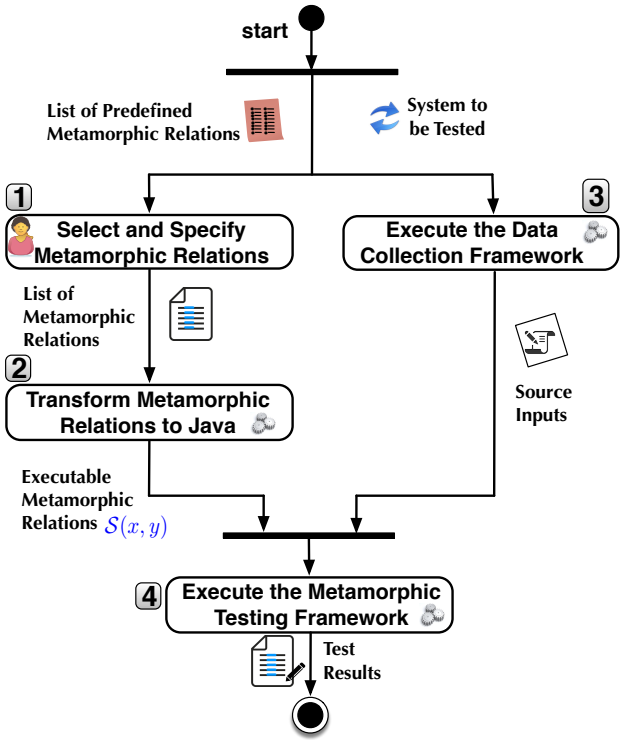


Figure 1: Overview of MST.

In Step 3, the engineer executes a Web crawler to automatically collect information about the system under test (e.g., URLs that can be visited by an anonymous user). The crawler determines the structure of the system under test and the actions that trigger the generation of new content on a page. The collected information includes source inputs for MST. To collect additional information, the engineer can process manually implemented test scripts, if available. Step 3 does not depend on other steps. Step 3 is presented in Section 2.2.5.

In Step 4, MST automatically loads the source inputs required by the MR and generates follow-up inputs as described by the relation. After the source and follow-up inputs are executed, their execution results are checked according to the MR. The details of the step are described in Section 2.2.6.

[11] The original MST toolset is available at `https://sntsvv.github.io/SMRL/`

OTG_AUTHZ_002_.smrl

```
1 import static smrl.mr.language.Operations.*
2 import smrl.mr.language.Action;
3
4 package owasp {
5   MR OTG_AUTHZ_002 {
6    {
7     for ( Action action : Input(1).actions() ){
8       IMPLIES(
9         cannotReachThroughGUI( User(2), action.url )                //1st
10           && !isSupervisorOf( User(2), action.user )                //par
11           && !isError(Output( Input(1),action.position) )           //of
12           && EQUAL( Input(2), changeCredentials(Input(1), User(2)) )//IMPLIES
13         ,
14           NOT( Output(Input(1),action.position).equals(   //2nd par of
15               Output(Input(2),action.position) ) )        //IMPLIES
16       ); //end-IMPLIES
17     } //end-for
18   }} //end-MR
19 }//end-package
```

Figure 2: An MR for the Bypass Authorization Schema vulnerability.

The MST DSL and the data collection framework can be extended to support new language constructs and data collection methods. The MST framework can be extended to deal with input interfaces that are not supported yet (e.g., Silverlight plug-ins [111]) and to load data collected by new data collection methods.

The following sections explain the details of each step in Fig. 1, with a focus on how MST achieves its automation objectives.

2.2.3 SMRL: A DSL for Metamorphic Relations

The MST approach starts with the activity of selecting and specifying MRs (Step 1 in Fig. 1). To enable specifying new MRs, the MST approach is supported by a DSL called Security Metamorphic Relation Language (SMRL). Engineers can also select MRs for the system under test from the set of predefined MRs (see Section 2.2.7).

SMRL is an extension of Xbase [57], an expression language provided by Xtext [22]. Xbase specifications can be translated to Java programs and compiled into executable Java bytecode. MST relies on Xbase since DSLs extending Xbase inherit the syntax of a Java-like expression language as well as language infrastructure components, including a parser, a linker, a compiler and an interpreter [57]. These features will facilitate the adoption of SMRL.

SMRL extends Xbase by introducing (i) a set of data representation functions, (ii) a set of boolean operators to specify security properties, and (iii) a set of Web-specific functions to express data properties and to transform data. These functions can also be extended by defining new Java APIs to be invoked in MRs.

Fig. 2 presents an MR written in the SMRL editor. The relation checks whether the URLs dedicated to specific users can be accessed by other users through a direct request. We use it as a running example in the rest of this Section.

In the following, we introduce the SMRL grammar, the boolean operators, the data representation functions, and the Web-specific functions.

SMRL Grammar The SMRL grammar extends the Xbase grammar, which extends the Java 8 grammar. Each SMRL specification can have an arbitrary number of import declarations which indicate the APIs to be used in MRs (Line 1 in Fig. 2).

Table 1: Excerpt of the data functions in SMRL.

Data function	Description
Input(int i)	Returns the i^{th} input sequence.
Action(int i)	Returns the i^{th} input action.
Session(int i)	Returns the i^{th} Web session.
User(int i)	Returns the i^{th} user of the system.
Output(Input i)	Returns the sequence of outputs generated by Input *i*.
Output(Input i, int n)	Returns the output generated by the n^{th} action of Input *i*.
HttpMethod()	Returns the name of an HTTP method (e.g., DELETE).
RandomFilePath()	Returns a file system path. We select paths of files in the Web system subfolder, ignoring images, and replacing symbolic links (e.g., 'plugins' is mapped to 'plugin' in Jenkins).
RandomValue(Type t)	Returns a random value of the given type.

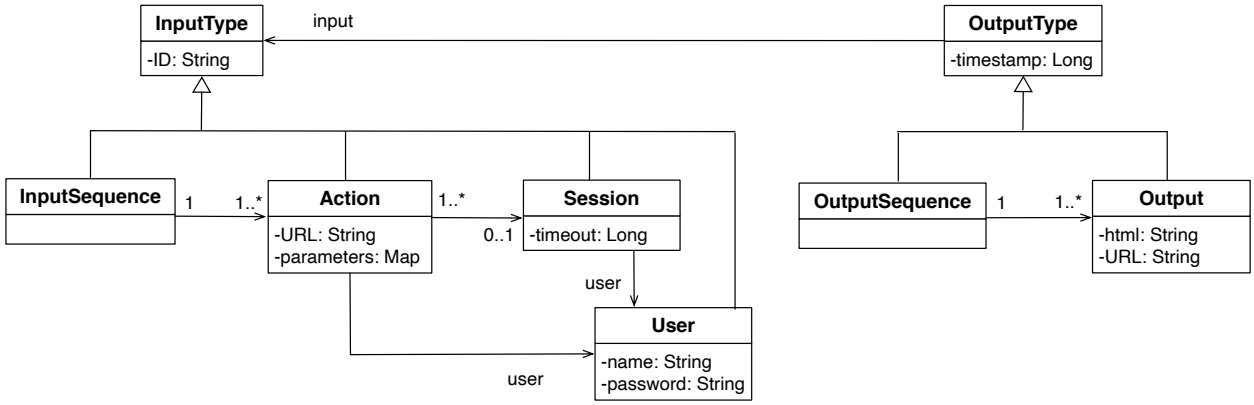


Figure 3: Metamorphic data classes in SMRL.

A package declaration resembles the Java package structure and can contain one or more MRs. Line 4 in Fig. 2 declares the package *owasp*, which is is the package for MST MRs. Like in Java, MRs defined in different SMRL specification files can belong to the same package.

An MR can contain an arbitrary number of `XBlock- Expressions`, which are nonterminal symbols defined in the Xbase grammar. An `XBlockExpression` can contain loops, function calls, operators, and other `XBlockExpressions`.

Data Representation Functions SMRL provides 18 functions to represent different types of data (i.e., system inputs and outputs) in MRs. Data is typically represented by a keyword followed by an index number used to identify different data items. To keep SMRL simple, data are represented with functions (hereafter *data functions*) with capitalized names (e.g., `Input(1)`). Table 1 presents a subset of the data functions in SMRL.

Each data function returns a data class instance. Fig. 3 presents the SMRL data model where all classes are subtypes of either `InputType` or `OutputType`. `InputType` represents input data that can be defined to trigger a certain system behavior. `InputSequence` represents a sequence of interactions between a user and the system under test and is consequently associated with `Action`. `Action` represents an activity performed by a user (e.g., requesting a URL). It carries information about actions such as a URL requested by an action and parameters in the URL query string. `Action` is associated with `Session`, which represents a user session in a Web application; `User` represents a system user.

A *source input* is an instance of `InputType` returned by one of the data functions; a *follow-up* input is an instance of `InputType` modified by means of a Web-specific function (see Section 2.2.3). For example, a source input might be a sequence of two HTTP requests for user login and user profile visualization. A follow-up input is the same sequence with login credentials for a different user. Instances of `OutputType` capture outputs generated by the system when processing an input; each instance of `OutputType` is associated with an

Table 2: Excerpt of the Web-specific functions in SMRL.

Operator	Description
changeCredentials(Input i, User u)	Creates a copy of the provided input sequence where the credentials of the specified user are used (e.g., within login actions).
copyActionTo(Input i, int from, int to)	Creates a new input sequence where an action is duplicated in the specified position and the remaining actions are shifted by one.
cannotReachThroughGUI(User u, String URL)	Returns true if a URL cannot be reached by the given user by exploring the user interface of the system (e.g., by traversing anchors).
isLogin(Action a)	Returns true if the action performs a login.
isSupervisorOf(User a,User b)	Returns true if 'a' can access the URLs of 'b'.
afterLogin(Action a)	Returns true if the action follows a login.
isSignup(Action a)	Returns true if the action registers a new user on the system.
isError(Output page)	Returns true if the page contains an error message.
userCanRetrieveContent(User u, Object out)	Returns true if the output data (i.e., the argument 'out') has ever been received in response to any of the input sequences executed by the given user during data collection.

instance of `InputType`. The last three functions in Table 1 return predefined/random values. They are used to redefine attributes of follow-up inputs as described in Section 2.2.6.

Boolean Operators SMRL provides seven boolean operators, i.e., `IMPLIES`, `AND`, `OR`, `TRUE`, `FALSE`, `NOT` and `EQUAL`. They enable the definition of *metamorphic expressions*, which are boolean expressions that should hold for an MR to be true. A *metamorphic expressions* is a specific kind of `XBlockExpression`. Metamorphic expressions are used to decompose an MR into simple properties. They are defined in a declarative manner, which is standard practice in MT [135].

The MR in Fig. 2 includes a metamorphic expression using the operator `IMPLIES`. Since the expression is within a loop body, the relation holds only if the expression evaluates to true in all the iterations over the input actions.

The semantics of operators `IMPLIES`, `AND`, `OR`, `TRUE`, `FALSE`, and `NOT` is straightforward. Operator `EQUAL`, instead, does not evaluate the equality of two arguments but defines a follow-up input by assigning the second parameter to the first parameter. Operator `EQUAL` acts as an equality operator only when its first parameter refers to an input that has already been used in previous expressions of the MR. Otherwise, it acts as an assignment operator. In Fig. 2, operator `EQUAL` defines the follow-up input `Input(2)` as a modified copy of `Input(1)`.

Web-Specific Functions MRs for security testing often capture complex properties of Web systems that cannot be expressed with simple boolean or arithmetic operators. Therefore, SMRL provides a set of functions that capture typical properties of Web systems and that alter Web data. Table 2 describes a portion of the 30 Web-specific functions in SMRL [26]. Each function is provided as a method of the SMRL API. Engineers can specify additional functions as Java methods. The new functions can be used in SMRL thanks to the underlying Xtext framework.

The MR in Fig. 2 uses the Web-specific functions `cannotReachThroughGUI`, `isSupervisorOf`, `isError` and `changeCredentials`. The relation indicates that the same sequence of actions should provide different outputs when performed by two different users under a condition. The condition is that one of the two users cannot access one of the requested URLs by browsing the GUI of the system. In other words, if the system does not provide a URL to a user through its GUI, then the user should not be allowed to access the URL. Also, to avoid false alarms, the user who cannot access the URL from the GUI, indicated as `User(2)` in Fig. 2, should not be a supervisor with access to all the resources of the other user, i.e., `User(1)`. Finally, the MR avoids source inputs that return an error message to `User(1)` because, for these inputs, it is not possible to characterize the output that should be observed for `User(2)`, who, indeed, may observe the same error, a different error, or an empty page.

In Fig. 2, function `cannotReachThroughGUI` checks if the URL of the current action cannot be reached from the GUI (Line 9). Function `isSupervisorOf` checks if `User(2)` is not a supervisor of `User(1)` (Line

OTG_AUTHZ_002.java

```
1  package owasp;
2  import smrl.mr.language.Action;
5  public class OTG_AUTHZ_002 extends MR{
6    public boolean mr() {
7      for (final Action action : Input(1).actions()) {
8        {
9          ifThenBlock();
10         if (((( cannotReachThroughGUI( User(2), action.getUrl())
11                 && (!isSupervisorOf(User(2), action.getUser())))
12                 && (!isError(Output(Input(1), action.getPosition())))
13                 && EQUAL(Input(2), changeCredentials(Input(1), User(2))))) {
14           ifThenBlock();
15           boolean _NOT = NOT( Output( Input(1), action.getPosition()).equals(
16             Output( Input(2), action.getPosition())));
17           if (_NOT) {
18             expressionPass(); /* //PROPERTY HOLDS" */
19           } else {
20             return Boolean.valueOf(false);
21           }
22         } else {
23           expressionPass(); /* //PROPERTY HOLDS" */
24     }}}
25     return true;
26 }}
```

Figure 4: Java code generated from the MR in Fig. 2.

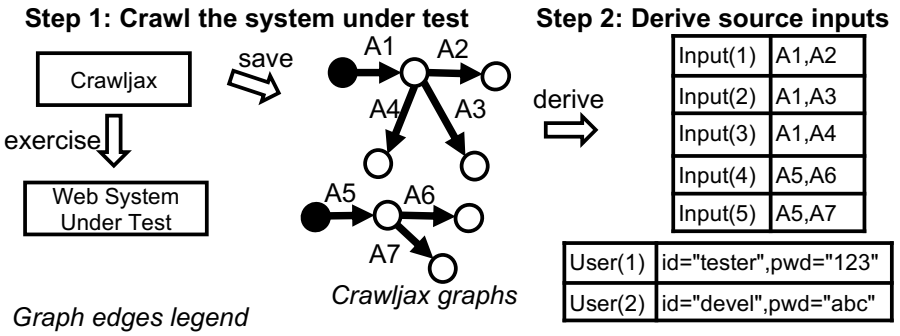
10). Function `isError` returns true if an output page contains an error message, based on a configurable regular expression (Line 11). Function `changeCredentials` creates a copy of a provided input sequence using different credentials. It is invoked to define the follow-up input (Line 12). Data function `Output` executes the sequence of actions in an input sequence (e.g., requests a sequence of URLs) and returns the output of the i^{th} action.

2.2.4 SMRL to Java transformation

SMRL specifications are automatically transformed into Java code (Step 2 in Fig. 1). To this end, MST relies on an extension of the Xbase compiler (hereafter SMRL compiler). Each MR is transformed into a Java class with the name of the relation and its package. The generated classes extend class `MR` and implement its method `mr`.

Method `mr` executes the metamorphic expressions in the MR. It returns `true` if the relation holds and `false` otherwise. To do so, the SMRL compiler transforms each boolean operator into a set of nested `IF` conditions. For example, for operator `IMPLIES`, the generated code returns `false` when the first parameter is true and the second one is false. For the case in which the MR holds, the SMRL compiler generates a statement that returns `true` at the end of `mr`.

Fig. 4 shows the Java code generated from the relation in Fig. 2. A loop control structure is generated from the loop instruction in the relation (Line 7). The loop body contains the Java code generated from the metamorphic expression using operator `IMPLIES` (Lines 10-24). The first `IF` condition checks whether the first parameter of operator `IMPLIES` holds (Lines 10-13). The nested `IF` block checks whether the second parameter of `IMPLIES` holds (Line 17). If the expression does not hold, `mr` returns `false` (Line 20). The relation holds only if all the expressions in the loop hold. Therefore, the SMRL compiler generates a `return true` statement after the loop body (Line 25). Calls to the methods `ifThenBlock` and `expressionPass` are used to erase the generated follow-up inputs at each iteration.



Action ID	Action Type	Element	URL	data
A1	CLICK	DIV[1]/BUTTON[1]	me.com/login	id="tester"; pwd="123"
A2	CLICK	DIV[2]/TABLE[1]/A[1]	me.com/stats	session={...}
A3	CLICK	DIV[2]/TABLE[1]/A[2]	me.com/startSlave	session={...}
A4	CLICK	DIV[2]/TABLE[1]/A[3]	me.com/profile	session={...}
A5	CLICK	DIV[1]/BUTTON[1]	me.com/login	id="devel"; pwd="abc"
A6	CLICK	DIV[2]/TABLE[1]/A[1]	me.com/stats	session={...}
A7	CLICK	DIV[2]/TABLE[1]/A[2]	me.com/profile	session={...}

Figure 5: Data collection with a simplified example.

2.2.5 Data Collection

To automatically derive source inputs (Step 3 in Fig. 1), MST relies on an extension of the Crawljax Web crawler [108, 107]. Crawljax explores the user interface of a Web system (e.g., by requesting URLs in HTML anchors or by entering text in HTML forms). It generates a graph whose nodes represent the system states reached through the user interface, and whose edges capture the action performed to reach a given state (e.g., clicking on a button). Crawljax detects the system states based on the content of the displayed page. Our extension relies on the edit distance to distinguish the system states [94]. We keep a cache of the HTML page associated to each state detected by Crawljax. When a new page is loaded, our extension computes the edit distance between the loaded page and all the pages associated to the different system states. When the distance is below a given threshold (5% of the page length), we assume that two pages belong to the same state. If a page does not belong to any state, Crawljax adds a new state to the graph. Crawling stops when no more states are encountered, or when a timeout is reached.

The MST Crawljax extensions enable replicating and modifying portions of a crawling session. In addition to (i) the Crawljax actions and (ii) the XPath of the elements targeted by the actions (e.g., a button clicked on), our extension records (iii) the URLs requested by the actions, (iv) the data in the HTML forms, and (v) the background URL requests. These additional data enable, for example, replicating modified portions of crawling sessions that request URLs not appearing in the last Web page returned by the system. To crawl the system under test, the MST extended Crawljax requires only its URL and a list of credentials.

Fig. 5 exemplifies the data collection steps. First, Crawljax generates the graphs of the system under test. Second, we automatically derive source inputs from the graphs. For example, an input sequence is a path from the root to a leaf of a Crawljax graph in a depth-first traversal. Third, the SMRL functions query the source inputs (see Section 2.2.6). For example, `Input(i)` returns the i^{th} input sequence; `User(i)` returns the i^{th} unique login credentials in the input sequences.

In addition to Crawljax, MST also processes manually implemented test scripts to generate additional source inputs. It processes test scripts based on the Selenium framework [20] and derives a source input from each script. MST relies on test scripts to exercise complex interaction sequences not triggered by Crawljax. Crawl-

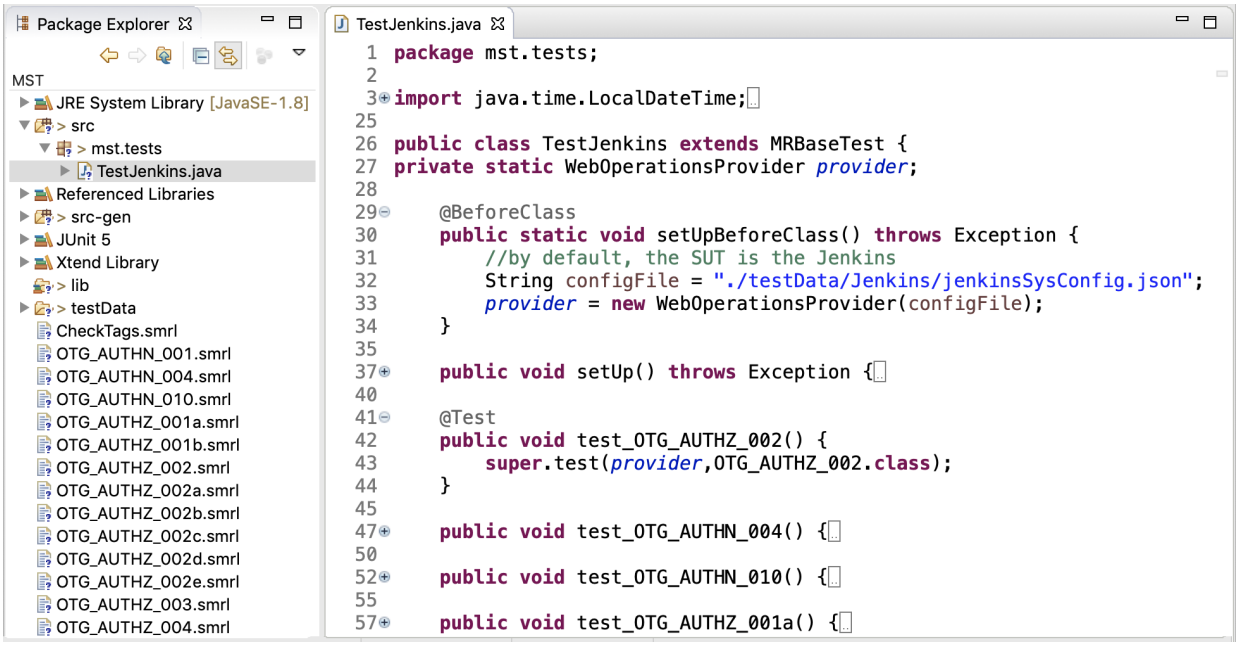


Figure 6: An example JUnit test case to select and execute MRs.

jax, instead, performs an almost exhaustive exploration of the Web interface, which is typically not done by test scripts. Engineers can reuse scripts developed for functional testing or define new ones.

2.2.6 Execution of Metamorphic Relations

MST automatically performs testing based on the executable MRs in Java and the data collected by the data collection framework (Step 4 in Fig. 1). The MST testing framework relies on the JUnit framework [16] to integrate MT into traditional testing environments. To select MRs to be executed, the engineer needs to write a JUnit test case. Fig. 6 presents a JUnit test case selecting multiple MRs (e.g., OTG_AUTHZ_002, OTG_AUTHN_004, and OTG_AUTHN_010) through function `test`. MRs need to be copied in the workspace (see the files with extension `.smrl` in the project explorer window in Fig. 6) and referred to in the JUnit test case (see TestJenkins.java in Fig. 6). MST provides a Java class, `MRBaseTest` (Line 26 in Fig. 6), which extends the JUnit framework with utility functions to facilitate the selection of MRs. Method `setUpBeforeClass` is used to specify the configuration for `WebOperationsProvider` (Lines 30-34), i.e., the component in charge of loading source inputs.

MRs are executed as standard JUnit [16] test classes through the Eclipse [14] user interface or the console wrapper. Each MR is treated as a distinct JUnit test case (Lines 42-57 in Fig. 6). For each MR, class `MRBaseTest` automatically invokes our MT algorithm that executes the MR.

Fig. 7 presents the MT algorithm used by MST. The algorithm takes as input an MR and a data provider exposing the collected data (source inputs). The MT algorithm first processes the bytecode of the MR to identify the types of source inputs referenced by the relation (e.g., *Input* and *User*). To do so, function `extractSourceInputTypes` (Line 2) identifies the calls to the *data representation functions* using the ASM static analysis framework [1]. The MT algorithm ensures that each source input is used in at least one execution and that all possible source input combinations are stressed during the execution of the relation (e.g., all available URLs with all configured users). This is achieved by function `iterateOverInputTypes` (Line 3). The function iterates over all available items for a given input type (e.g., all available users) and is recursively invoked for each input type in the MR.

```
Require: MR, the bytecode of the metamorphic relation to be executed
Require: dataProvider, an object that exposes the data collected by the crawlers
Ensure: Failures, a list of failing executions with contextual information
 1: function EXECUTEMETAMORPHICTESTING(MR, dataProvider)
 2:     srcTypes ← extractSourceInputTypes(MR)
 3:     iterateOverInputTypes(MR, dataProvider, 0, dataTypes)
 4:     return Failures
 5: end function
 6: function ITERATEOVERINPUTTYPES(MR, dataProvider, i, dataTypes)
 7:     while dataProvider.hasMoreViews(dataTypes[i]) do
 8:         dataProvider.nextView(dataTypes[i])
 9:         if (i < dataTypes.lenght) then   //need to iterate over other types
10:             iterateOverInputTypes(MR,dataProvider, i+1,srcTypes)
11:         else  //we have set a view for every input type in the relation
12:             result = MR.run() //execute the metamorphic relation
13:             if ( result == false) //the MR does not hold
14:                 addFailure(Failures,dataProvider) //trace the failure
15:         end if
16:     end while
17: end function
```

Figure 7: Metamorphic testing algorithm.

Function `iterateOverInputTypes` is driven by the methods exposed by the data provider (Lines 7 and 8). The data provider works as a circular array that provides, in each iteration of `iterateOverInputTypes`, a different view on the collected data. For N input items of a given type (e.g., User), function `nextView` (Line 8) generates N different views with items shifted by one position.

After the views are generated, the MR is executed (Line 12). The algorithm generates follow-up inputs from the source inputs at each invocation of operator `EQUAL`. For example, in Fig. 2, operator `EQUAL` makes `Input(2)` refer to a copy of the input sequence returned by function `changeCredentials`.

If the MR does not hold (Lines 13 and 14), function `addFailure` stores the failure context information (i.e., source-inputs, follow-up inputs, and system outputs). MST reports only failures that perform HTTP requests (e.g., accessing a URL) not generated by input sequences that led to previously reported failures. Therefore, it reduces the time spent to manually analyze failures triggered by distinct follow-up inputs exercising the same vulnerability.

Function `nextView` is iteratively invoked until all the items of a given input type are processed (Line 7). This guarantees that all input item combinations are used. For the data functions providing random values, `nextView` returns 100 different views by default. Since this may lead to combinatorial explosion, we test each MR for a maximum of 24 hours.

Fig. 8 exemplifies the execution of the relation in Fig. 2. The table on the left represents the sequence of functions invoked by our algorithm. In this example, two views for `User` are inspected for each view of `Input`. The first two invocations of `MR.run` return true (not shown in Fig. 8) because the *login* and *stats* pages have been accessed by both users *devel* and *tester* and thus the implication holds. The third invocation of `MR.run` returns false because the output page for the *startSlave* URL is the same for the two input sequences and thus the relation does not hold. To determine if Web pages are equal, we rely on edit distance.

2.2.7 Catalog of Metamorphic Relations

The original MST work [98] includes a catalog of MRs that has been derived from the activities described in the OWASP book on security testing [109]. The book provides detailed descriptions of 90 testing activities (hereafter *OWASP security testing activities*) for Web systems; each OWASP security testing activity targets a specific vulnerability. For example, for the bypass authorization schema vulnerability, OWASP suggests to collect links in administrative interfaces and to directly access the corresponding URLs by using credentials of other users. Based on this suggestion, the MR in Fig. 2 has been defined.

Since we can perform some of the OWASP security testing activities in multiple ways (e.g, we may discover bypass authorization vulnerabilities by directly accessing a reserved URL not provided by the GUI or by

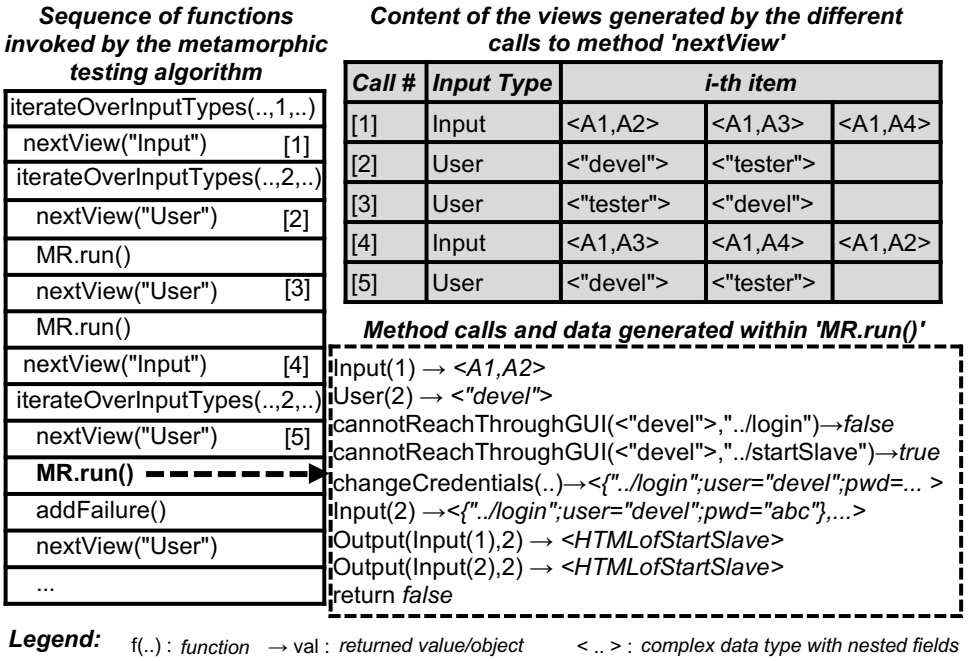


Figure 8: Example data processing for the relation in Fig. 2.

accessing a URL after changing some additional URL parameter values), we have more than one relation for each of those activities. Also, not all the activities mentioned in the OWASP book benefit from MT (for example, some of these activities require inspection of source code). The MST catalog includes 22 MRs which automate 16 OWASP activities. Table 3 provides the list of MRs in our catalog grouped according to the OWASP testing activity they aim to automate.

The relations in our catalog rely on the following observation. We perform security testing using follow-up inputs that cannot be generated by interacting with the GUI of the system but conform with the input format of the system and match its configuration (e.g., the URLs requested by the unauthorized user refer to existing system resources). MST inherits from mutational fuzzing [157] the idea of generating follow-up inputs by altering valid source inputs. However, to generate valid inputs that match the system configuration, MST does not rely on random values. Instead, MST modifies source inputs using the data provided by the SMRL Web-specific functions, which return domain-specific information (e.g., protocol names) and crawled data. Finally, by capturing properties of the output generated by the source and follow-up inputs, MST identifies vulnerabilities that cannot be detected with implicit oracles.

Table 4 presents an excerpt from the MST catalog with a description of each MR. The entire catalog is available for download [26]. All the MRs are expressed using an implication (operator `IMPLIES`). Operator `EQUAL` is used to define follow-up inputs. It indicates that the follow-up input (typically `Input(2)`) is a copy of the source input (usually `Input(1)`) except for the differences made by the function calls following the operator. For example, in `OTG_AUTHN_001`, the follow-up input is equal to the source input except for one action of the input sequence which should be performed on the HTTP channel.

All the MRs include a loop, which enables defining multiple follow-up inputs by iteratively modifying different actions of the source input. For example, `OTG_AUTHN_001` works with all the login actions observed in the source input sequence. Function `isLogin()` returns true only if the current action performs a login; otherwise, the implication trivially holds, and no follow-up input is generated.

In the MST catalog, the right-hand side of the implication usually captures the relation between the outputs of the source and follow-up inputs. For instance, according to `OTG_AUTHN_001`, the output for the follow-up input (which performs a login on the unencrypted HTTP channel) should be different from the output for the source input because it should not be possible to login using the HTTP channel.

Table 3: Catalog of MRs provided with MST

OWASP testing activity	MR ID
Testing for Credentials Transported over an Encrypted Channel	OTG_AUTHN_001
Testing for Bypassing Authentication Schema	OTG_AUTHN_004
Testing for Weaker Authentication in Alternative Channel	OTG_AUTHN_010
Testing Directory traversal/file include	OTG_AUTHZ_001a and OTG_AUTHZ_001b
Testing for Bypassing Authorization Schema	OTG_AUTHZ_002, OTG_AUTHZ_002a, OTG_AUTHZ_002b, OTG_AUTHZ_002c, OTG_AUTHZ_002d, and OTG_AUTHZ_002e
Testing for Privilege Escalation	OTG_AUTHZ_003
Testing for Insecure Direct Object References	OTG_AUTHZ_004
Test Number of Times a Function Can be Used Limits	OTG_BUSLOGIC_005
Test HTTP Strict Transport Security	OTG_CONFIG_007
Testing for Weak Encryption	OTG_CRYPST_004
Testing for HTTP Verb Tampering	OTG_INPVAL_003
Testing for HTTP Parameter pollution	OTG_INPVAL_004
Testing for Session Fixation	OTG_SESS_003
Testing for Logout Functionality	OTG_SESS_006
Test Session Timeout	OTG_SESS_007
Testing for Session puzzling	OTG_SESS_008

2.2.8 MST-related work

MST aims to address the limitations of security testing approaches. Indeed, MST supports oracle automation thanks to MRs that can precisely capture the relations between inputs and outputs. Considerable research has been devoted to developing MT approaches for various domains such as computer graphics (e.g., [106, 66, 81, 89]), simulation (e.g., [48, 55, 118]), Web services (e.g., [41, 139, 158]), embedded systems (e.g., [145, 39, 88, 77]), compilers (e.g., [141, 91]), variability and decision support (e.g.,[134, 133, 129, 87]), bioinformatics (e.g., [46, 124, 125]), numerical programs (e.g., [45, 24]), and machine learning (e.g., [152, 117]). For instance, Chat et al. [40, 41] present an MT approach for Service-Oriented Applications (SOA). Their approach encapsulates the services under test, execute source and follow–up test cases, and check their results. Segura et al. [134, 133] provide a test data generation technique using MRs for feature model analysis tools. The technique automatically generates test cases using stepwise transformations that ensure that MRs hold at each step.

Although there is considerable research devoted to MT, very little attention has been paid to its application to security testing [132]. Preliminary applications of MT to security testing [43] focus on the functional testing of security components (i.e., verifying the output of code obfuscators and the rendering of login interfaces) and the verification of low-level properties broken by specific security bugs (e.g., the heartbleed bug [140] which affects the relation between the size of the payload data field of an SSL message and the length declared in the same message). Although these works show the feasibility of MT for security testing, they focus on a narrow set of vulnerabilities and do not automate the generation of executable metamorphic test cases, which are manually implemented based on the identified MRs. In contrary, MST supports the specification of MRs for various vulnerabilities and automates the generation of executable metamorphic test cases from the MRs.

Although MT is highly automatable, a high number of MT works in the literature have the main contribution as a case study on the application of MT to specific testing problems [132]. For instance, Kuo et al. [88] report a case study on the use of metamorphic testing for the detection of faults in a wireless metering system. Ding et al. [55] present a case study on the detection of faults in a Monte Carlo modeling program for the simulation of photon propagation. Chen et al. [45] present yet another case study on the application of MT to programs implementing partial differential equations. These works do not provide any systematic method to specify MRs with proper tool support. Very few approaches provide proper tool support enabling engineers to write system-level MRs [132]. These few approaches require that MRs be defined either as Java methods [159] or pre-/post-conditions [116], which limit the adoption of MT to verify system-level security properties. Furthermore, since MRs often employ a declarative notation, the use of an imperative language may force engineers to invest additional effort to translate abstract, declarative MRs. To avoid the additional effort, we propose our

Table 4: Excerpt of the MST metamorphic relation catalog for security testing.

OTG-AUTHN-001: Testing for credentials transported over an encrypted channel

```
MR OTG_AUTHN_001 {
 {
    for ( Action action : Input(1).actions() ) {
        var pos = action.getPosition();
        IMPLIES(
            isLogin(action)                                                  //1st par (1st clause)
              &&  EQUAL ( Input(2) , Input(1) )                              //1st par (2nd clause)
              &&  Input(2).actions.get(action.position).setChannel("http")   //1st par (3rd clause)
            ,
            different ( Output(Input(1),pos),  Output(Input(2),pos) )        //2nd par of IMPLIES
        );//end-IMPLIES
    }//end-for
 }
}//end-MR
```

Description: A login operation should not succeed if performed on the http channel. The 1st parameter of the operator `IMPLIES` is a boolean expression with three clauses joined with logical conjunctions. The 1st clause checks if the current action performs a login. The 2nd clause defines the follow-up input. The 3rd clause changes the channel of the login action in the follow-up input. The 2nd parameter of IMPLIES checks if the output generated by the login operation is different in the two cases.

OTG-AUTHZ-001: Testing for directory traversal/file include

```
MR OTG_AUTHZ_001 {
 {
    for ( Action action : Input(1).actions() ){
        for ( var par=0; par < action.getParameters().size(); par++ ){
            var pos = action.getPosition();
            IMPLIES(
                EQUAL( Input(2), Input(1) ) //1st par of IMPLIES (1st clause)
                  && Input(2).actions().get(pos).setParameterValue(par, RandomFilePath())//(2nd clause)
                ,
                OR(     //2nd par of IMPLIES, OR operator receiving 2 parameters
                  isError( Output(Input(2),pos) )   //1st par of OR
                  ,
                  userCanRetrieveContent(action.getUser(), Output(Input(2),pos)) )//2nd par of OR
            );//end-IMPLIES
        }//end-for
    }//end-for
 }
}//end-MR
}
```

Description: A file path passed in a parameter should never enable a user to access data that is not provided by the user interface.
This metamorphic relation contains two nested loops; the first iterates over the actions in the input sequence, the second iterates over the parameters of the action. The 1st parameter of the operator `IMPLIES` is a boolean expression with two clauses joined with a logical conjunction. The 1st clause defines a follow-up input that is a copy of the source input. The 2nd clause set the value of a parameter to a random file path. The 2nd parameter of IMPLIES verifies the result. It is implemented as an OR operation where the 1st parameter verifies that the follow-up input leads to an error page. The 2nd parameter deals with the case in which the generated request is valid, and verifies that the returned content is something that the user has the right to access. The framework evaluates the MR as many times as needed to provide 100 different random file paths to the parameters of the action in the position `pos`.

OTG-SESS-003: Testing for session fixation

```
MR OTG_SESS_003 {
 {
    for( Action signup : Input(1).actions() ){
        for ( Action f : Input(2).actions() ) {
            var pos = f.getPosition();
            IMPLIES(
                isSignup(signup) && //1st par of IMPLIES (1st clause)
                afterLogin( f ) && //1st par of IMPLIES (2nd clause)
                EQUAL( Input(3), addAction( Input(2), pos+1, signup ))  //(3rd clause)
                ,
                different( Output(Input(3), pos).getSession(),  //2nd par of IMPLIES
                           Output(Input(3), pos+1).getSession())
            );//end-IMPLIES
        }//end-for
    }//end-for
 }
}//end-MR
```

Description: A signup action should always lead to a new session ID, even when performed by a user who is already logged-in. This
metamorphic relation contains two nested loops iterating over the actions of two distinct source input sequences (i.e., `Input(1)` and `Input(2)`). The first loop looks for a signup action (i.e., 'signup'), the second looks for an action (i.e., 'f') following a login. The 1st parameter of the operator `IMPLIES` is a boolean expression with three clauses joined with a logical conjunction. The 1st clause checks if we are in the presence of a signup action. The 2nd clause checks if the action 'f' follows a login. The 3rd clause defines a follow-up input by copying the signup action after the action 'f' in the source input `Input(2)`. The 2nd parameter of IMPLIES verifies the result by checking that the session ID following the signup action is different than the one of the previous page.

DSL as part of MST (see Section 2.2.3). Segura et al. [130, 131] propose a template-based approach for the description of MRs. The proposed template aims to ease communication among practitioners. It does not support security-related language constructs. There is also no automated support to transform template-based MRs into executable test cases.

2.3 Misuse Case Programming - MCP

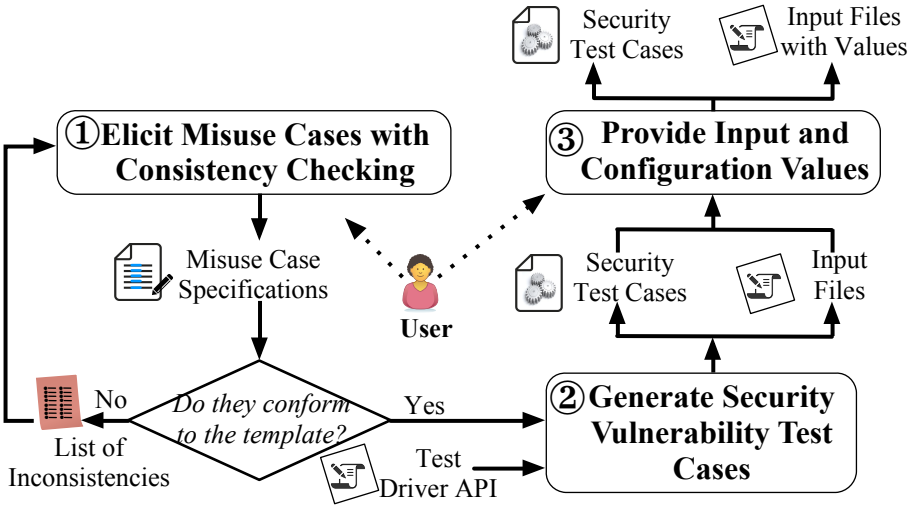


Figure 9: MCP Overview

Misuse Case Programming (MCP) is an approach developed by the University of Luxembourg to automatically generate security vulnerability test cases from misuse case specifications [99]; it is supported by an automated prototype toolset[12] [100]. Fig. 9 presents an overview of the approach. In Step 1, the user manually elicits misuse cases according to the Restricted Misuse Case Modeling (RMCM) template [97] that includes keywords to support the extraction of control flow information.

Once security requirements are captured in the form of misuse cases, MCP automatically checks whether the misuse case specifications conform to the RMCM template. If any inconsistency is detected, the MCP toolset reports these inconsistencies.

In Step 2, MCP processes the misuse case specifications once they are deemed consistent. It automatically generates executable test cases and test input files from the misuse case specifications and a test driver API that implements basic security testing activities (e.g., requesting a URL or starting a network sniffing tool). In Step 3, the user provides input values for the generated test input files to be used during testing. In the rest of this section, we elaborate on each step in Fig. 9.

2.3.1 Elicitation of Misuse Cases

The user elicits misuse cases based on the RMCM method. RMCM extends the Restricted Use Case Modeling (RUCM) method [155, 151, 69, 71, 70] to support the specification of security requirements in an analyzable form [97].

RMCM provides a template that characterizes basic and alternative flows in misuse cases. A basic threat flow describes a nominal scenario for a malicious actor to harm the system (Lines 4 - 13 in Fig. 10). It contains misuse case steps and a postcondition. Alternative flows capture failed attacks (e.g., Lines 21 - 26), while alternative *threat* flows describe alternative attack scenarios. For instance, in Lines 14 - 20, 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, specified by the `RFS` keyword. The `IF .. THEN` keyword describes the conditions under which alternative flows are taken (e.g., Line 16). The RMCM keywords are used to specify actors (e.g., `MALICIOUS user`), successful attacks (e.g., `EXPLOIT`), attack patterns (e.g., `PROVIDE SQLI VALUES`) and control flow (e.g., `FOREACH`). All other terms in a specification are freely selected by the user.

[12]The original MCP toolset is available at `https://sntsvv.github.io/MCP/`

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 a 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 Flow (SAF1)**
22 RFS 6
23 1. IF the response page contains an error message THEN
24 2. RESUME STEP 7.
25 3. ENDIF.
26 **Postcondition** The malicious user cannot access the resource dedicated to users with a different role.

Figure 10: 'Bypass Authorization Schema' misuse case specification.

MCP is provided with a set of predefined misuse case specifications derived from the OWASP testing guidelines [109]. They can be reused (or adapted) across projects; in addition, the user can write new, system specific misuse cases.

2.3.2 Automated Generation of Security Vulnerability Test Cases

MCP takes as input a test driver API and misuse case specifications to automatically generate security vulnerability test cases. To generate test cases, MCP borrows some concepts from *natural language programming*, referring to techniques automatically producing software programs (e.g., executable test cases) from specifications in Natural Language (NL) [27, 123]. More precisely, MCP includes an initial Natural Language Processing (NLP) step where it automatically derives, from each misuse case specification, a misuse case model that captures the control flow of the activities described in the specification. As part of natural language programming, it translates the derived models into sequences of executable instructions (e.g., invocations of the test driver API's functions) that implement malicious activities. To do so, MCP adapts the idea, developed by other works [103, 92, 68, 90], of combining similarity measures and ontologies [65]. Similar to other natural language programming techniques, MCP automatically builds an ontology that captures the structure of the given test driver API. It generates executable instructions by looking for nodes in the ontology similar to phrases in misuse case steps.

The process in Fig. 11 presents an overview of the MCP approach for automated generation of executable test cases. 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 includes a generic test driver API for Web testing that is extendable for system-specific operations.

MCP generates as output a set of executable security test cases that rely on the provided test driver API to perform the malicious 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 (i) to model the target programming language and test infrastructure concepts (Label A in Fig. 11), (ii) to capture the relationships and structure of the classes of the

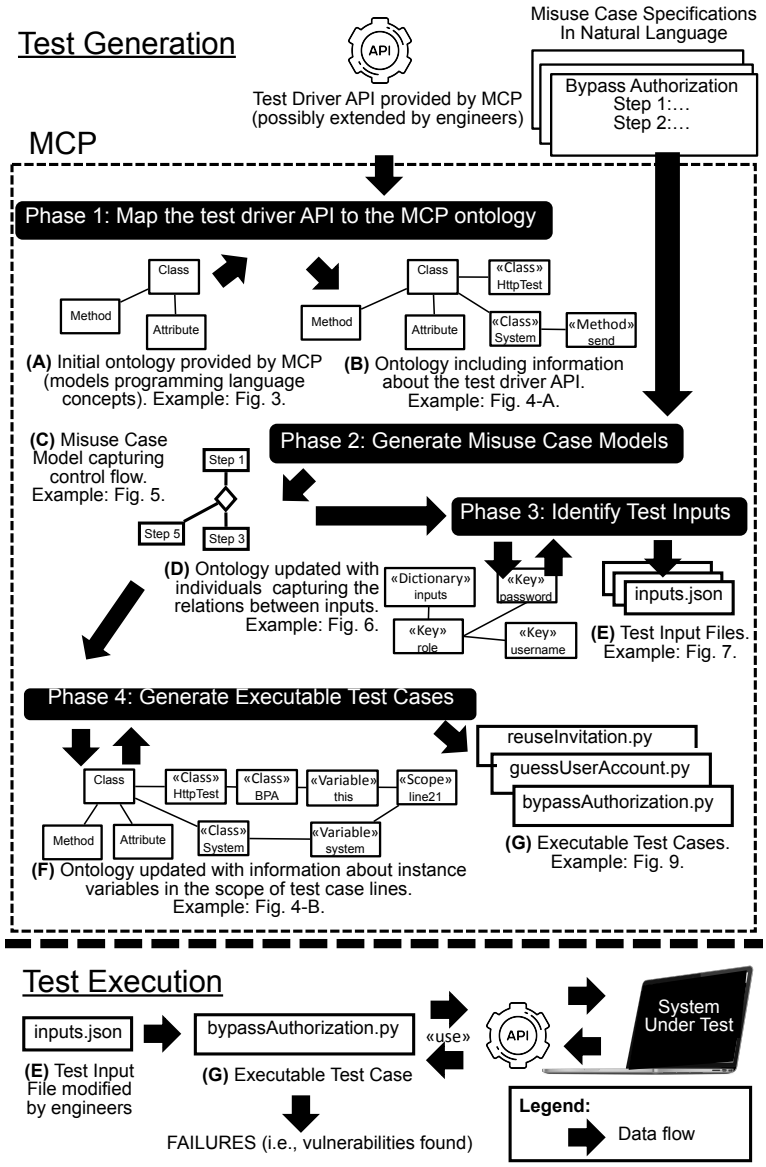


Figure 11: Automated generation of executable test cases with MCP.

test driver API (Label B), (iii) to capture the relationships between inputs (Label D), and (iv) to represent the variables declared in the generated test case (Label F). We employ an OWL ontology [21] because it provides simple means to query the modeled data.

To generate security vulnerability test cases from misuse case specifications, MCP works in four phases. In Phase 1, *Map the test driver API to the MCP ontology*, MCP processes the test driver API and augments the MCP ontology with the individuals representing the classes and functions in the given test driver API (Labels A and B). In Phase 2, *Generate misuse case models*, MCP uses an NLP pipeline to derive models that explicitly capture the control flow implicitly described in misuse case specifications (Label C).

In Phase 3, *Identify test inputs*, MCP determines the inputs to be sent to the system. It first identifies the input entities and augments the MCP ontology with individuals capturing relations between inputs (Label D). It then generates a configuration file filled in by engineers with input values to be used during test execution (Label E). MCP can automatically generate input values when these values can be derived based on predefined strategies (e.g., grammars to derive inputs for code injection attacks [144, 23]).

In Phase 4, *Generate executable test cases*, MCP automatically generates executable test cases from the misuse case models (Labels C and G). Each test case follows the control flow in the corresponding misuse case model.

For each step in the model, it 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. More specifically, this solution maps NL commands (i.e., sentences in misuse case steps) to objects and methods in the provided API by retrieving information from the MCP ontology (Label F). Mapping is based on string distance and identification of synonyms (i.e., MCP similarity measures). MCP augments the ontology with individuals matching the variables declared in the test case.

MCP automatically determines test input entities, input relationships, and values to be assigned to input entities. It expects that input entities appear in misuse case steps with a verb in which one or more entities are sent to the system by an actor (e.g., *The malicious user provides an anomalous weight to the system*). It employs Semantic Role Labeling (SRL) [78], which automatically determines the roles played by words in a sentence, to identify misuse case steps that describe the action of sending an entity to a destination. For test oracles, MCP automatically generates source code from misuse case condition steps checking erroneous outputs and from misuse case steps indicating that the malicious user can exploit a vulnerability.

MCP generates, for each misuse case, an executable test case, a JSON test input file without input values (see Fig. 12), and some configuration files for the test driver API (see Fig. 13).

Configuration files are used to configure general purpose test API methods for a specific system. For Web systems, these methods send inputs to the system through a Web page. MCP automatically generates configuration files for these methods.

Each generated test case is a Python class implementing a `run` method executing the malicious activities described in a given misuse case. Fig. 14 shows part of the test case generated from the misuse case in Fig. 10. MCP declares and initializes the variables `system`, `maliciousUser` and `inputs` (Lines 3 - 5). The variable `system` refers to an instance of the class `System` whose methods trigger the functions of the system under test (e.g., `request`). The variable `maliciousUser` refers to the test class simulating the malicious user behavior. The variable `inputs` refers to a dictionary populated with input values in the JSON input file. The instructions in the test case (e.g., a call to an API method) are selected based on string similarity between the phrases in the misuse case steps and the variables, methods and parameters of the test driver API.

2.3.3 Providing Input and Configuration Values

The user provides input values for the generated JSON input file. The generated input file reflects the relations between input entities in the misuse case (see Fig. 12). Fig. 15 shows example input values for the JSON input file in Fig. 12.

Simple inputs are represented as key-value pairs; complex inputs are given as a list of dictionaries (e.g., `role` in Fig. 12). A complex input is generated every time an input entity (e.g., `role`) is referred to in an iteration (i.e., a step containing the keyword `FOREACH`) because the body of iterations typically contains activities that provide a set of related inputs to the system (e.g., `username` and `password`). Every input entity referred to in the body of the iteration is captured by a key-value pair (e.g., the case for the key `username`). Nested

```
{"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-EXECUTABLE-CODE",
      "the_resource_contains_the_role_parameter_in_the_HTTP_post_data": "PUT...
    },
     ADD-MORE-ENTRIES
    ],
  },
  ADD-MORE-ENTRIES
  ]
}
```

Figure 12: Input file generated by MCP.

```
"URL" : "http://link_to_the_page",
"Position" : "Choose POST-data, URL or BOTH",
"username" : "USERNAME_id_in_the_page",
"password" : "PASSWORD_id_in_the_page",
"Others" : ["list_of_other_parameters"]
```

Figure 13: Configuration file generated to send inputs through to the login page.

bypassAuthorizationSchema

```
class bypassAuthorizationSchema(HTTPTester):
    def run(self):
        system = System(path=self.rootPath)
        maliciousUser = self
        input = self.loadComplexInput("inputs.json")
        roleIter = input["role"].__iter__()
        while True:
            try:
                role = roleIter.__next__()
                parameters = dict()
                parameters["password"] = role["password"]
                parameters["username"] = role["username"]
                system.send("login page", parameters)
                resourceIter = role["resource"].__iter__()
                while True:
                    try:
                        resource = resourceIter.__next__()
                        if not eval(resource["the_resource_contains_the_role_parameter_in_the_URL"]):
                            if not eval(resource["the_resource_contains_the_role_parameter_in_the_HTTP_post_data"]):
                                system.request(resource)
                                maliciousUser.responsePage = system.responsePage
                                parameter = dict()
                                parameter["error_message"] = resource["error_message"]
                                if not maliciousUser.responsePage.contain(parameter):
                                    parameters = dict()
                                    parameters["resource"] = resource["resource"]
                                    parameters["role"] = role["role"]
                                    system.exploit(parameters)
                                else:
                                    maliciousUser.abort("The MALICIOUS user CANNOT execute a function dedicated ...")
```

Figure 14: Part of test case automatically generated from misuse case specification 'Bypass Authorization Schema' in Table 10.

iterations are captured by nested lists of dictionaries (e.g., the case of `resources`), since they usually describe groups of activities working on additional sets of related inputs. A special type of input are the values evaluated in conditional statements in specifications (e.g., the case of `the_resource_contains...`) as, for these values, the user can provide either a boolean value (e.g., `TRUE`) or a Python expression to be evaluated at runtime.

Configuration files are generated to match input entities in misuse cases and input parameters of Web pages of the system (see Fig. 13). Fig.16 shows the values provided to the configuration file in Fig. 13.

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

Figure 15: Part of the JSON file in Fig. 12 with input values.

```
"URL" : "http://www.icare247.eu/?q=micare_user_login",
"Position" : "POST-data",
"password" : "passwd",
"username" : "email",
"Others" : ["op", "form_build_id", "form_id"]
```

Figure 16: Values provided to the configuration file in Fig.13

The test case is executed using the Python interpreter. During execution, the test case loads the inputs and configuration values from the JSON files and invokes the test driver API functions when needed. Vulnerabilities are reported by the test driver API method `exploit` (Line 28 in Fig. 14).

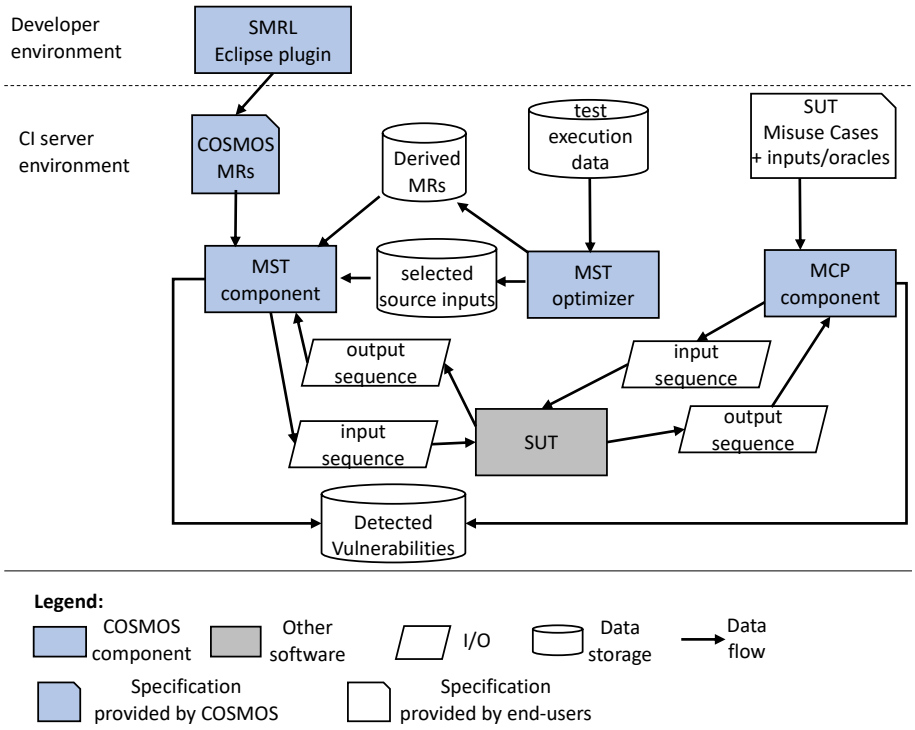


Figure 17: COSMOS security testing solution.

3 COSMOS Security Testing Toolset Architecture

In this Section we recall the architecture of the COSMOS Security Testing Toolset, which has been described in deliverable D2.1. The COSMOS Security Testing Toolset architecture is depicted in Figure 17.

The COSMOS security testing solution consists of three components that automate (1) metamorphic security testing (MST component), (2) misuse case programming (MCP component), and (3) metamorphic testing optimization (MST optimizer). Their interactions are shown in Figure 17. As further shown in Figure 17, they perform security testing of the software under test (SUT) by automatically generating inputs and by verifying outputs at the system level. They shall report the list of vulnerabilities detected. In Figure 17 we also show that MRs can be implemented by a software engineer within its development environment using the MST SMRL Eclipse plugin[13].

In COSMOS, MST is used to discover unknown vulnerabilities by relying on MRs covering a large set of vulnerability types (COSMOS MRs in Figure 17), including CPS-specific ones. This report mainly focus on the extensions to MST developed during the reported period. Precisely, the COSMOS consortium has focussed on the definition of an extended catalog of MRs (described in Section 4.2) and has extended the MST toolset to include all the features required to implement the extended catalog of MRs (described in Section 4.1).

Although MST addresses the oracle problem (see Section 2.2.2) and automates the identification of test inputs (it relies on a Web crawler for source inputs and automated generation of follow-up inputs), it might be inefficient. The metamorphic testing optimizer, which is part of future COSMOS developments, will rely on machine learning to improve MST efficiency. First, it will derive source inputs based on execution data collected during other testing campaigns (e.g., functional) or in the field. Second, it will determine the subset of source inputs to be used during metamorphic testing; for example, it may rely on reinforcement learning to determine, based on historical data, which source inputs enable the discovery of a vulnerability (e.g., the ones

[13]The SMRL Eclipse plugin is available for download at `https://github.com/SNTSVV/SMRL_EclipsePlugin/`

covering diverse instruction sets or the ones already successful in previous executions). Third, it will rely on the automated identification of properties of execution data to derive new metamorphic relations.

MCP is a technique that automatically generates executable security test cases mimicking security attacks. It processes attack descriptions in natural language, thus enabling all the stakeholders to understand the attack specifications. However, differently from MST, it requires manual specification of inputs and oracles. For this reason, within the COSMOS project, which puts a large emphasis on testing automation, MCP will be used only to automate testing for documented software misuses not covered by MST. The COSMOS extensions to MCP concern known limitations of the tool and are described in Section 5.

4 Metamorphic Security Testing: Extended MST Component

COSMOS developed an extension of the MST toolset that enables the identification of a large proportion of the vulnerability types that are reported in the MITRE Common Weaknesses Enumeration catalog [3] and concern Web systems (i.e., systems with a user interface that can be accessed through a Web-browser). COSMOS has focused on Web systems because they are often used to access and control CPSs. An example is the COSMOS use case *SOMATOM go* scanner system provided by the Siemens (see COSMOS deliverable D2.1), which is configured through a Web interface usually configured through a private LAN. Another example is Webmin, which is a web-based interface for system administration for Unix. Webmin is extensively used for setting up Ubuntu [146] and, in general, OS derived from Debian [61], such as Raspian for Rasperry PI[14] [67]. For example, INTELLIGENTIA, one of the COSMOS partners, uses Webmin extensively for its internal IT resources; in general, Webmin is widely adopted in industry to manage embedded OSs.

In the following, we describe our extended MST toolset (Section 4.1) and discuss the empirical assessment performed to determine the extent to which MST can discover known vulnerability types, which led to the definition of an extended catalog of MRs (Section 4.2). Finally, in Section 4.3 we report about a preliminary study concerning the application of MST to the automotive context.

4.1 Extended MST Features

In this section we describe the functionalities introduced into the MST toolset to enable the definition of the extended catalog of MRs described in Section 4.2. Detailed information about the vulnerability types discovered by our extended catalog of MRs are provided in Appendix A.

Our extension for the MST toolset does not concern the grammar of the Security Metamorphic Relation Language (SMRL, see Section 2.2.3) but the supporting functions and the framework to execute MRs; in other words, our changes concerned not the SMRL Eclipse plugin but what was previously referred to as *SMRL library*, which in COSMOS is simply referred to as *MST component*. The extended SMRL library includes 10950 lines of code, 66 Java classes, and 1118 methods.

In the following, we provide a description of the main extensions performed, grouped into macro-categories.

OS support The extended MST framework can be executed both on Windows and on Unix-like (Linux and OsX) operating systems.

Browser support The extended MST framework can perform testing of a Web system using either Google Chrome or Mozilla Firefox.

Support for stateful MRs The extended MST framework provides both stateless (default) and stateful MRs. In a stateless MR the state of the browser used to provide inputs to the SUT is reset after every input sequence (i.e., a source input or a follow-up input). In a stateful MR the browser is not reset. Stateful MRs are required when a source input sequence should be split into multiple follow-up input sequences because the output of former sequences is used to modify latter ones. The engineer specifies which portion of the MR shall be stateful. The stateful portion of a MR is delimited by the function calls *setResetBrowserBetweenInputs(true)* and *setResetBrowserBetweenInputs(false)*.

An example of a stateful MR is CWE_610_384, which verifies if the SUT authenticates a user without first invalidating the existing session (see Figure 18). Within CWE_610_384, the source input is split into two

[14]Raspberry Pi is a series of small, single-board computer systems that is used for the development of CPS (e.g., weather monitoring and robotics)

```
58 MR CWE_610_384 {
59         {
60             for ( var x = 0; x < Input(1).actions().size(); x++ ){                                                    //(1)
61
62                 for ( var y = x+1; isLogin(Input(1).actions().get(x)) && (y < Input(1).actions().size()); y++){       //(2)
63
64                     setResetBrowserBetweenInputs( false );                                                            //(3)
65
66                     IMPLIES (
67                         EQUAL ( Input(2) , Input ( Input(1).actions().subList(0, y ) ) ) &&                           //(4)
68                         EQUAL ( Input(3) , Input ( Input(1).actions().subList(y, Input(1).actions().size() ) ) )  && //(5)
69                         Input(3).addAction(0,Input(1).actions().get(x)) &&                                            //(6)
70                         Input(3).actions().get(1).setSession( Output(Input(2),x).getSession() )                       //(7)
71                         ,
72                         AND( different ( Output(Input(2),x).getSession(),  Output(Input(3),0).getSession())  //
73                             , isError(Output(Input(3),1)))              //(8)
74
75                     ); //end-IMPLIES
76
77                     setResetBrowserBetweenInputs( true );
78
79                 } //end-for
80             } //end-for
81         }
82     } //end-MR
83 } //end-package
```

Figure 18: MR CWE_610_384.

follow-up inputs, the former (hereafter, *F1*) executes all the actions until a login action (defined in Line 67), the latter (hereafter, *F2*) executes the following actions (defined in Line 68). A new login action is added at the beginning of F2 (Line 69). Then, we simulate the activity of a malicious user who has stolen (or guessed) the session ID generated after the first login action (hereafter, old session ID) and set it into his current browser session[15], after the second login (Line 70). Ideally, the old session should be invalid and prevent the malicious user to retrieve content (i.e., the original content is not retrieved or an error message is displayed). Please note that we do not perform the login with another user because the session should be invalidated independently from the user who is accessing the system. The two follow-up inputs are then executed one after the other and (Line 72) the MR verifies that the session ID after the last action of F1(i.e., after the first login) is different from the session ID after the first action of F2 (i.e., after the second login); moreover, the output retrieved after performing the action with the old session ID should be an error (Line 73).

Support for catalogs Many vulnerabilities can be discovered through attacks based on catalogs, that is, by testing the SUT repeatedly with inputs (usually strings) belonging to a specific set of inputs; such set of inputs is usually referred to as attack payload list. An example is given by Cross-Site Scripting (*XXS*) vulnerabilities, which can be exploited by inserting into form inputs crafted strings such as `<SCRIPT>alert('XSS');</SCRIPT>`.

We provide 11 catalogs that provide the following attack strings:

- SQL Injection Strings
- CRLF Attack strings
- Code Injection strings
- Static Injection strings
- Weak passwords
- Special characters
- XSS injection strings
- LDAP injection strings
- Command injection strings
- XQuery injection strings
- Cron expressions injection strings

[15]Since it is not possible to know which cookie is the session cookie, MST simply copies all the cookies provided by the SUT.

```
MR CWE_79c_storedXSS {
{
    //Dialogs are normally ignored by our framework by clicking on OK; with the following we avoid clicking on OK
    keepDialogsOpen = true;                                                                          //(1)

    for (Action action : Input(1).actions()){                                                        //(2)
        var pos = action.getPosition();                                                              //(3)

        //We try every form input
        for (var x = 0; action.containFormInput() && x < action.formInputs.size; x++){               //(4)

            IMPLIES(
                //We are about to submit a form
                action.eventType == Action.ActionType.click &&                                       //(5)
                //We did not test the URL before
                notTried( x+action.url, Input(1).actions().get(pos).getElementURL() ) &&             //(6)
                //No alert shown normally
                ! Output(Input(1),pos).hasAlert                 &&
                //We create the follow-up input
                EQUAL(Input(2), Input(1) )                      &&                                   //(7)

                //We put an XSS payload into a form input text
                Input(2).actions.get(pos).setFormInput(x, XSSInjectionString()) &&                   //(8)

                //We will perform the same sequence as the source-input (i.e., without XSS injected)
                //If an alert will pop-up, it will mean that the XSS has been stored into the system
                EQUAL(Input(3), Input(1) )                                                           //(9)
                ,
                OR(                                                                                  //(10)
                     //either the attack was not performed
                     Output(Input(2), pos).emptyFile,
                     //or no effect shall be observed (the effect of the XSS is usually visualized when reaching the page where we inject the XSS)
                     ! Output(Input(3), pos-1).hasAlert        //stored XSS
                )
            );
        }
    }

 } }
}
```

Figure 19: MR CWE_79c_storedXSS.

Each string in the provided catalogs is treated as a source input by the MST toolset, which means that in MRs they are referred to by using a dedicated data representation function (e.g, *XSSInjectionString()*). An example is shown in Figure 19 (MR named *CWE_79c_storedXSS*), where an XSS injection string is provided to a form input and then MST verifies if, after accessing the same page again in a different follow-up input (i.e., *Input(3)*), a popup is shown). CWE_79c_storedXSS enabled us to replicate the vulnerability CVE-2020-8820 affecting Webmin 1.941 and earlier.

In addition, the extended MST toolset provides also catalogs with entries pointing to files with invalid types and files containing XML injections.

Finally, the extended MST toolset enables an engineer to rely on custom catalogs; this is achieved by invoking, within a MR, the data representation function *PayloadEntry(catalogName)*, which loads, one after the other, all the attack strings provided by the specified catalog. A catalog shall be provided as a text file whose path shall be specified in the MST json configuration file.

Support for additional Web-specific functions We introduced the function *reservedKeywords(User)* which returns a list of words that are observed only in the output for a given user; it is used to determine if a malicious user can access sensitive information for another user.

We introduced the function *isResetPassword(Action)*, which is used to determine if the action concerns resetting the password for a user.

We introduced the Action *RequestUrlAction(String url)*, which requests a URL specified (statically or dynamically) in the MR (e.g., to generate a follow-up input where the same URL is requested twice). For example, Figure 20 shows the MR for CWE 352 which verifies that a user shouldn't be able to successfully submit a form without retrieving it first; otherwise, someone else can create a crafted page with a submit form button. In this case function *RequestUrlAction* is used to load a version of the form that has been stored on the local filesystem and then submitting it within a new user session.

Support for remote data collection We have implemented the data representation function *RemoteFile(String location)*, which represents a file stored on the system where the SUT is running. The file to be

```
29 MR CWE_352 {
30     {
31
32     keepDialogsOpen = true;
33
34     for (Action action : Input(1).actions()){                                    //(1)
35         var pos = action.getPosition();                                          //(2)
36
37
38         for (var x = 0; action.containFormInput() && x < action.formInputs.size; x++){
39
40
41             IMPLIES(
42                 notTried( x+action.url, Input(1).actions().get(pos).getElementURL() ) && //(5)
43                 afterLogin( action )    &&                                        //(6)
44                 EQUAL( Input(2), Input( LoginAction(action.user)   ,              //(7)
45                 RequestUrlAction("file://"+Output(Input(1),pos-1).file.absolutePath),
46                         action
47                     ) )
48                                                 //(9)
49                 ,
50                 different ( Output(Input(1),pos),  Output(Input(2),3) )
51
52             );
53         }
54     }
55
56  } }
57 }
```

Figure 20: MR CWE_352

loaded is specified by indicating absolute location of the file. It provides functions that return the lines of the file or the lines added after the last access to the file within a MR.

We also implemented the data representation function named *Log*, which is used to refer to the different log files accessible on the system where the SUT is running. It is used, for example, in the MR CWE_359_313_532_538 shown in Figure 21, which verifies that accessing a resource dedicated to the user performing the action does not make the system log information reserved to the user performing the action (e.g, password as returned by function *reservedKeywords*).

Tool output. To facilitate adoption it is extremely important to provide information that facilitates the understanding of MR failures. After every failure the extended MST reports:

- the list of source inputs used by the MR (for each input it print all its actions);
- the list of follow-up inputs used by the MR (for each input it print all its actions);
- the list of actions within source inputs and follow-up inputs whose outputs had been processed by the MR; they provide information about the action leading to an unexpected result (i.e., the follow-up action) and the expected result (which can be derived from the source input and the MR).

The information described above is reported both on the console and as failure information within the Eclipse JUnit plugin (when MRs are executed on the developer's laptop). When MRs are executed remotely, the information is collected by the CI/CD framework as for standard JUnit test cases.

An example failure for the bypass authorization schema MR (see Figure 22) is shown in Figure 23.

The console output in Figure 23 below provides the following information:

- The failure was detected when testing the MR OTG_AUTHZ_002 (see JUnit tab).
- The inputs used when the failure was observed are: Input(2), which is a follow-up input, and Input(1), which is a source input. They differ for the user performing the action (i.e., user1 VS admin), as specified in line 34 of the MR in Figure 22.

```
27  * Idea:
28  * when we identify a page that can be accessed only by a user then it means that the information contained in it is
29  * reserved, thus the log generated when processing it should not contain reserved information.
30  *
31  *
32  *
33  * Comments:
34  * (1) For loop iterates over all actions of the Input(1).
35  * (2) Creates a cache of the current status of the log file.
36  * (3) Verified that current action follows a login.
37  * (4) Verifies that the current action is reserved to the current user (i.e., a follow-up user cannot perform it)
38  * (5) Verifies that the current URL leads to some output
39  * (6) Verifies that the that new lines generated into the log file when processing the input (6a)
40  *      do not include keywords that appear only in pages of the user and in particular in the page generated for the current action (6b).
41  *
42  *
43  */
44 MR CWE_359_313_532_538 {
45  {
46
47     for (Action action : Input(1).actions()){                                    //(1)
48
49             var initialLog = Log();                                              //(2)
50
51
52             IMPLIES(
53                 afterLogin(action) &&                                            //(3)
54                 cannotReachThroughGUI(User(),action.url)    &&                   //(4)
55                 Output(Input(1),action.position).values.size > 0                 //(5)
56                 ,
57
58                 NOT ( containsAny(
59                         Log().newLines(initialLog),                              //(6a)
60                         Output(Input(1),action.position).values.filter[ e | reservedKeywords(action.user).contains(e) ].toList() //(6b)
61                 ) )
62
63             );
64         }
65     }
66
67  }
68 }
```

Figure 21: MR CWE_359_313_532_538

- The execution of the MR led to the collection of output information for Input(1) and Input(2) in the following order:
  - Input(1); indeed, it is requested in line 37 of the MR
  - Input(1); indeed, it is the first input requested in line 38 of the MR
  - Input(2); indeed, it is the second input requested in line 38 of the MR
- For all the inputs above, the action verified by the 'Output' call is the third Action (i.e., the one that accesses the URL *http://192.168.56.102:8080/computer/slave1/launchSlaveAgent*). It means that the failure is observed when verifying the output for the fourth action (i.e., action number 3, actions are enumerated counting from 0). This happens within the iteration in Line 30.

The information above enables the end-user to understand the problem as follows: an unauthorized user (i.e., *(user1,user1Pass)*) can access a URL he should not (i.e., the URL *http://192.168.56.102:8080/computer/slave1/launchSlaveAgent* what is accessed by action number 3, the fourth action). This is what characterizes a real vulnerability affecting Jenkis (see `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999004`). Please note that the same information is also provided within the JUnit *Failure Trace* box on the left.

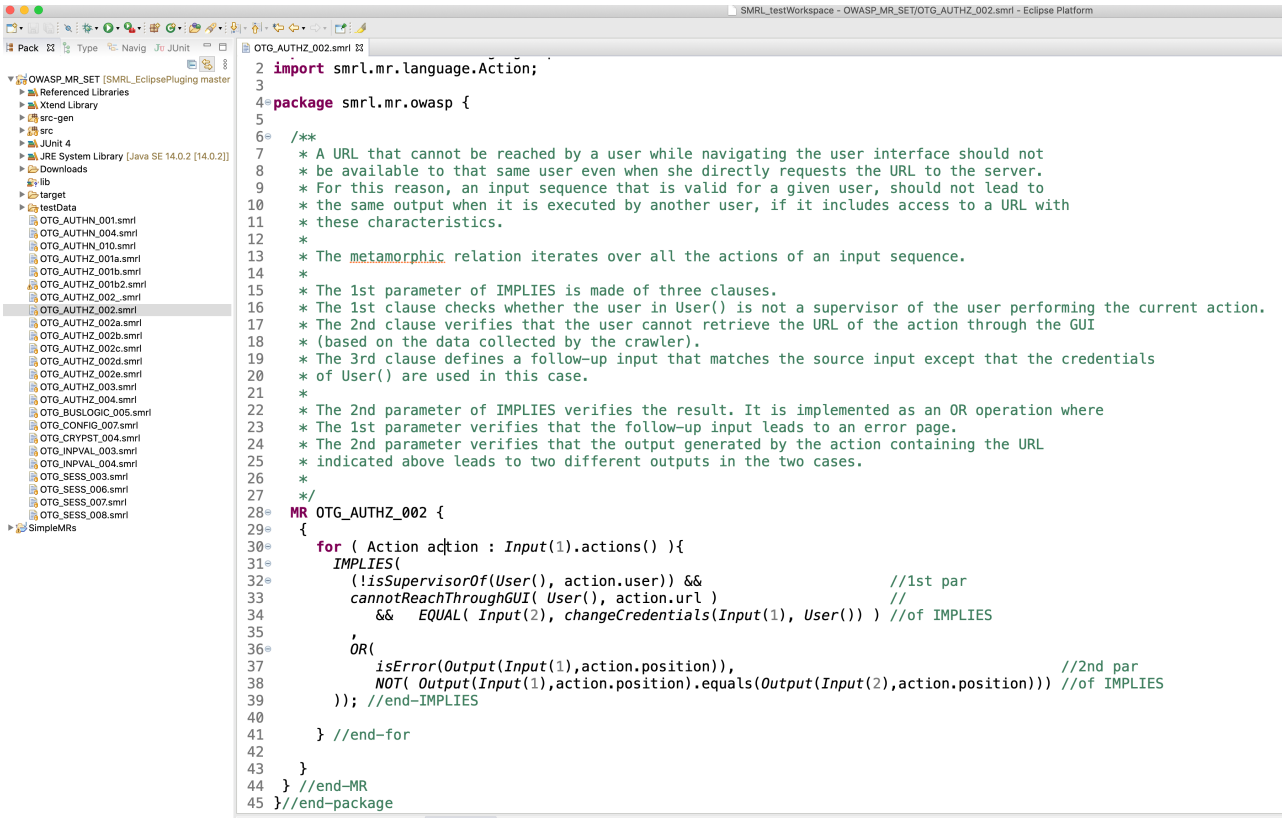


Figure 22: Bypass Authorization Schema MR OTG_AUTHZ_002.

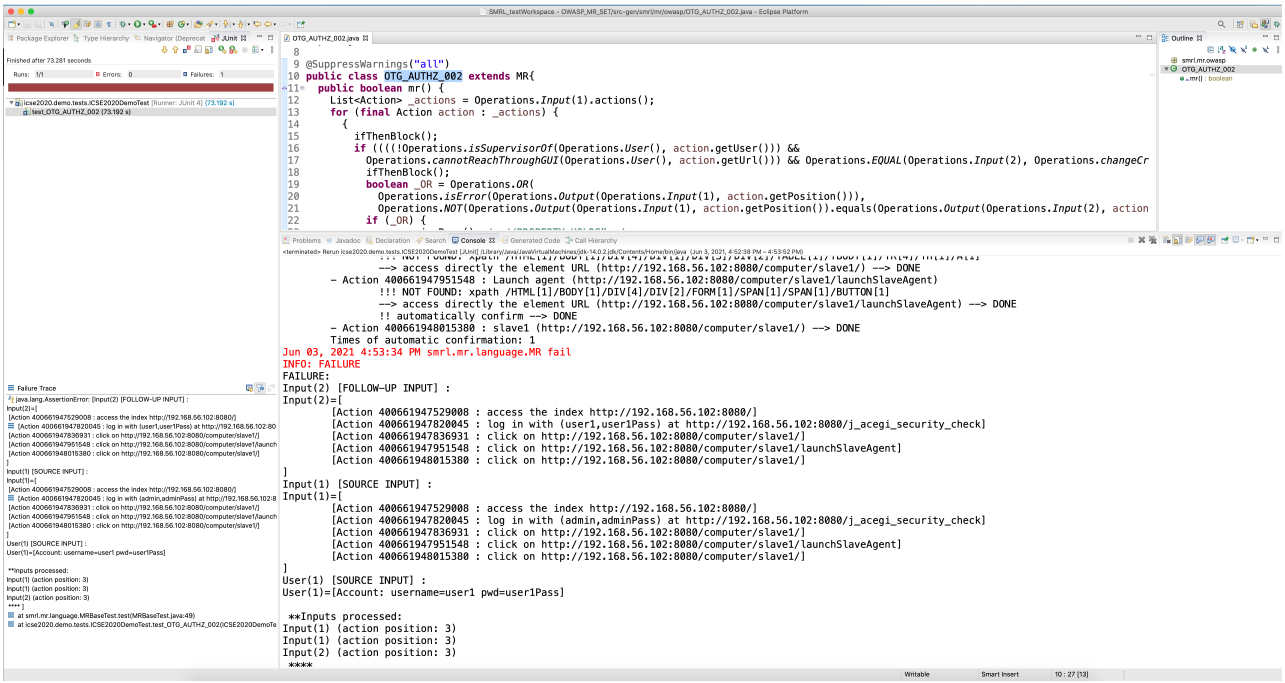


Figure 23: Failure for Bypass Authorization Schema MR 22; it detects Jenkins vulnerability *CVE-2018-1999004*.

4.2 Definition of an extended catalog of MRs

This section provides an overview of the extended catalog of MRs that has been defined by COSMOS in order to investigate the following Research Question (RQ1):

- ***RQ1. To what extent is metamorphic testing applicable in the context of security testing?*** MST has been designed and implemented to perform security testing by reasoning on outputs of multiple executions of the system under test. Not every type of vulnerability can be discovered through a relationship between outputs of multiple test executions (e.g., some may require program analysis). This research question aims to determine, in a systematic way, the types of security vulnerabilities that can and cannot be addressed by MST.

4.2.1 Subjects of the Study

To address our research question in a systematic way, we studied the list of weaknesses reported in the Common Weakness Enumeration (CWE) database [3]. We provide the following definitions of *vulnerability* and *weakness* since their definitions in the CWE framework [7] lack clarity[16]. A *vulnerability* is a specific fault of the system under test that causes the system to breach its security requirements. A *weakness* represents a fault type (i.e., the type of a vulnerability). It describes a human error made in the analysis, design, or implementation of the system that may affect the degree to which the system meets its security requirements.

The CWE database is organized into distinct views, each view grouping weaknesses according to a different set of categories, which are *common security architectural tactics* [9], *software development concepts* [12], *research concepts* [11], *software fault patterns* [6], *most dangerous errors* [8], and *hardware design* [10]. Other views map the weaknesses to some security-related catalogs (e.g., OWASP Top 10 [19] and SERT CEI C Coding standards [5]).

The CWE view for *common security architectural tactics* organizes weaknesses according to *security design principles*. This view has twelve categories representing the individual security design principles that are part of a secure-by-design approach to software development. It covers, in total, 223 weaknesses. The security design principles assist engineers in identifying potential mistakes that can be made when designing software [126, 127]. A weakness is thus the result of a design principle not being followed. For instance, the weaknesses in design principle *Authenticate Actors* are related to authentication-based components in the system. These components deal with verifying that the actor interacting with the system is whom this actor claims to be. The weaknesses in this category lead to a degradation of the quality of authentication if they are not addressed when designing and implementing the system under test [9]. The views for *software development concepts* and *hardware design* organize weaknesses based on the types of errors that affect the software implementation (e.g., illegal pointer dereferences) and the hardware design (e.g., faults in semiconductor logic), respectively. The views for *software fault patterns* and *research concepts* group implementation errors into categories capturing fault patterns [30] or high level descriptions of the faulty behaviour of the software (e.g., incorrect comparison and improper access control).

In our analysis, we focus on the weaknesses in the CWE view for common security architectural tactics [9], the weaknesses in the view for the CWE Top 25 most dangerous software errors (CWE Top 25) [8], and the weaknesses in the view for the OWASP Top 10 Web security risks (OWASP Top 10) [13]. We select the common security architectural tactics view because it enables us to determine the security design principles that MST can verify. We do not consider the view for *software development concepts* because MST is a black-box testing approach, that does not aim to discover specific implementation errors (e.g., type errors). We also

[16]The definitions provided by CWE are unclear since they rely on *synonyms* to distinguish vulnerability and weakness, as follows: 'Weaknesses are *flaws*, *faults*, *bugs*, and other *errors* in system design, architecture, code, or implementation that if left unaddressed could result in systems and networks, and hardware being vulnerable to attack. Weaknesses can lead to vulnerabilities. A vulnerability is a *mistake* in software or hardware that can be used by a malicious user to gain access to a system or network [7]'.

ignore the views for *software fault patterns*, *research concepts*, and mappings to *coding standards* [5] since they focus on software implementation. We ignore the CWE view for *hardware design* since MST does not address hardware vulnerabilities.

In our analysis, we include the CWE Top 25 and OWASP Top 10 views to assess to what extent MST can address the most widespread and critical security vulnerabilities. The CWE Top 25 view lists twenty-five most widespread weaknesses which are often easy to find and exploit. These weaknesses are considered dangerous because they often *allow attackers to completely take over the control of software, steal data, or prevent software from working [8].* The OWASP Top 10 [19] is the list of the ten most common web application security risks edited by the Open Web Application Security Project [18], i.e., an online community producing freely-available articles, methodologies, documentation, tools, and technologies in the field of web application security. It is updated every three to four years. The most up-to-date version includes 43 weaknesses grouped into 10 categories [13].

4.2.2 Analysis Procedure

To respond to RQ1, we compute, for each category in the views, the percentage of weaknesses that can be automatically discovered by MST.

Weaknesses are systematically analyzed with the objective of writing, for each one, one or more MRs using SMRL. For each weakness, we first inspect its description, its demonstrative examples, the description of concrete vulnerabilities (CVE) and common attack patterns (CAPEC) [2] associated with the weakness. Based on the information collected from our inspection, we implemented, using SMRL, a new MR that address the weakness or reused, if possible, an MR already available in the MST catalog. Each time we could not do so, we kept track of the reasons preventing the writing of an MR.

We report the percentage of the weaknesses that can be automatically discovered by MST. Since some of the weaknesses in the CWE database are specific to certain types of systems (e.g., Java Enterprise [15]), we identify weaknesses that refer to specific systems. We then distinguish between the results achieved with all the weaknesses (i.e., generic and specific), and the results achieved with the generic weaknesses only.

To better characterize the weaknesses that cannot be addressed by MST, we analyze the distribution of the reasons preventing the application of our approach, across the categories of the views considered in our analysis. Finally, we discuss the percentage of the weaknesses belonging to the CWE Top 25 and OWASP Top 10 lists.

To provide concrete examples of the CWE weaknesses that we have analyzed, we report, in Table 5, a subset of the weaknesses in the CWE view for common security architectural tactics. We refer to Table 5 in the rest of the section. Columns *Design principle* and *Weakness* report specific security design principles and names of weaknesses, respectively. Columns *Belongs to Top 25* and *Belongs to Top 10* indicate whether a weakness also belongs to the CWE Top 25 view or the OWASP Top 10 view, respectively. We also indicate if the weakness can be addressed by MST.

A preliminary investigation of this research question appeared in the Ph.D. Thesis of X. Phu Mai [101]; however, that study was based on the perceived feasibility of implementing MRs, based on their CWE descriptions. In practice, the author did not implement any MR to cover CWE descriptions; the implementation of MRs for CWEs has been entirely done within COSMOS.

Table 5: Subset of the security weaknesses in the CWE view for common security design principles.

Design principle	Weakness	Belongs to Top 25	Belongs to Top 10	Generic Weakness	Addressed by MT	Reason MST cannot be applied (R)
Audit	Omission of Security-relevant Information	No	Yes	Yes	No	R2
	Obscured Security-relevant Information by Alternate Name	No	No	Yes	No	R5
Authenticate Actors	Improper Authentication	Yes	Yes	Yes	Yes	TF3
	Weak Password Recovery Mechanism for Forgotten Password	No	Yes	Yes	No	R2
Authorize Actors	Improper Privilege Management	No	Yes	Yes	Yes	TF2
	Process Control	No	No	Yes	No	R1
Encrypt Data	Small Space of Random Values	No	No	Yes	No	R0, R2
	Missing Encryption of Sensitive Data	No	Yes	Yes	No	R6
Limit Access	Improper Restriction of XML External Entity Reference	Yes	Yes	Yes	Yes	TF2
	External Control of File Name or Path	No	Yes	Yes	Yes	
Manage User Sessions	J2EE Bad Practices: Non-serializable Object Stored in Session	No	Yes	No	No	R1
	Insufficient Session Expiration	No	Yes	No	Yes	
Validate Inputs	Cross-site Scripting	Yes	Yes	Yes	Yes	
	Deserialization of Untrusted Data	Yes	Yes	No	No	R2

```
MR CWE_79_a_XSSreflected {
 {
    //Dialogs are normally ignored by our framework by clicking on OK; with the following we avoid clicking on OK
    keepDialogsOpen = true;

    for ( Action action : Input(1).actions() ) {

        var pos = action.getPosition();

        //We try to inject every URL parameter, one after teh other
        for ( var x =0; x < action.parameters.size; x++){        //(2)

            IMPLIES(
                //We verify to not have tested the same URL before; there is no reason for testing a same URL
                 notTried( x, action.url )&&
                 //No pop-up / alert dialog shown normally
                ! Output(Input(1), pos).hasAlert   &&
                //We create the follow-up input as a copy of the source input
                EQUAL( Input(2), Input(1) ) &&                        //(4)
                //We put an XSS payload as parametr text
                Input(2).actions().get(pos).setParameterValue(x,XSSInjectionString())
                ,
                OR(
                    //either the attack was not performed
                    Output(Input(2), pos).emptyFile,
                    //or no effect shall be observed (the effect of the XSS is usually visualized when reaching the page where we inject the XSS)
                  ! Output(Input(2), pos).hasAlert  )
        );//end-IMPLIES
    }//end-for parameter
   }//end-for action
  }
}//end-MR
}//end-package
```

Figure 24: MR CWE_79a_XSSreflected.

```
MR CWE_79b_XSSreflected {
 {
    //Dialogs are normally ignored by our framework by clicking on OK; with the following we avoid clicking on OK
    keepDialogsOpen = true;                                                                    //(1)

    for (Action action : Input(1).actions()){                                                  //(2)
        var pos = action.getPosition();                                                        //(3)

        for (var x = 0; action.containFormInput() && x < action.formInputs.size; x++){         //(4)

            IMPLIES(
                //We are about to submit a form
                action.eventType == Action.ActionType.click &&                                //(5)
                //We did not test the URL before
                notTried( x+action.url, Input(1).actions().get(pos).getElementURL() )  &&      //(6)
                //No pop-up / alert dialog shown normally
                ! Output(Input(1),pos).hasAlert &&                                             //(7)
                //Create follow-up input
                EQUAL(Input(2), Input(1) )  &&                                                 //(8)
                //We put an XSS payload into a form input text
                Input(2).actions.get(pos).setFormInput(x, XSSInjectionString())                //(9)
                ,


                    OR(                                                                        //(10)
                     //either the attack was not performed
                     Output(Input(2), pos).emptyFile,
                     //or there is no pop-up in the output (reflected XSS immediately leads to a pop-up)
                     ! Output(Input(2), pos).hasAlert        //reflected XSS
                )
            );
        }
    }

 } }
}
```

Figure 25: MR CWE_79b_XSSreflected.

4.2.3 Results

Table 6 provides the list of the MRs that we implemented to discover vulnerabilities concerning security design principles; in total, we implemented 53 MRs. In total, after joining the newly developed MRs and the original catalog of MST MRs, COSMOS can rely on 60 distinct MRs to test Web-based systems.

In Table 6, each MR is named using the ID of the CWE weakness it covers. For well known vulnerability types (e.g., the ones triggered by means of code injections), we also add a representative keyword as suffix; this is the case for CWE_79a_XSSreflected. In case there are two possible ways to discover a same vulnerability type, we add suffix letters (i.e., 'a', 'b', 'c'); this is the case for CWE_79a_XSSreflected (Figure 24), CWE_79b_XSSreflected (Figure 25), and CWE_79c_XSSstored (Figure 26). In CWE_79a_XSSreflected we verify that the source input does not lead to pop-up windows, then we provide an XSS payload into a URL parameter and we verify that the output does not include a pop-up window (otherwise the system is vulnerable from reflected XSS, which means that an XSSString provided into an URL makes the browser of the user execute the script). In CWE_79b_XSSreflected we perform the same actions of CWE_79a_XSSreflected except that the XSS payload is provided into a text input that is passed into a form text input (they are usually passed as parametrs of a POST request not as URL parameters). In CWE_79c_XSSstored, the XSS payload is pro-

```
MR CWE_79c_storedXSS {
{
    //Dialogs are normally ignored by our framework by clicking on OK; with the following we avoid clicking on OK
    keepDialogsOpen = true;                                                                          //(1)

    for (Action action : Input(1).actions()){                                                        //(2)
        var pos = action.getPosition();                                                              //(3)

        //We try every form input
        for (var x = 0; action.containFormInput() && x < action.formInputs.size; x++){               //(4)

            IMPLIES(
                //We are about to submit a form
                action.eventType == Action.ActionType.click &&                                       //(5)
                //We did not test the URL before
                notTried( x+action.url, Input(1).actions().get(pos).getElementURL() ) &&             //(6)
                //No alert shown normally
                ! Output(Input(1),pos).hasAlert                 &&
                //We create the follow-up input
                EQUAL(Input(2), Input(1) )                      &&                                   //(7)

                //We put an XSS payload into a form input text
                Input(2).actions.get(pos).setFormInput(x, XSSInjectionString()) &&                   //(8)

                //We will perform the same sequence as the source-input (i.e., without XSS injected)
                //If an alert will pop-up, it will mean that the XSS has been stored into the system
                EQUAL(Input(3), Input(1) )                                                           //(9)
                ,
                OR(                                                                                  //(10)
                    //either the attack was not performed
                    Output(Input(2), pos).emptyFile,
                    //or no effect shall be observed (the effect of the XSS is usually visualized when reaching the page where we inject the XSS)
                    ! Output(Input(3), pos-1).hasAlert        //stored XSS
                )
            );
        }
    }

 } }
}
```

Figure 26: MR CWE_79c_storedXSS.

vided into a form text input but we do not expect the XSS effect to be shown right after the action; in this case we expect the XSS to be stored into the server and, therefore we access the page again to verify that it does not show the XSS pop-up. In case a CWE weakness has been already covered by the original catalog of MRs provided in past work about MST (see Table 3 on Page 18), we have added the corresponding MR ID as suffix (e.g., CWE_266_267_268_OTG_AUTHZ_002).

Most of our MRs can discover more than one weakness; this happens because different weaknesses characterize different types of faults, however, such faults may lead to the same failures and may be triggered using the same inputs. This is the case for CWE_266_267_268_OTG_AUTHZ_002, which uncovers the CWEs having the IDs 266, 267, and 268. The definition for CWE 266 is *A product incorrectly assigns a privilege to a particular actor, creating an unintended sphere of control for that actor*. The definition for CWE 267 is *A particular privilege, role, capability, or right can be used to perform unsafe actions that were not intended, even when it is assigned to the correct entity.* The definition for CWE 268 is *Two distinct privileges, roles, capabilities, or rights can be combined in a way that allows an entity to perform unsafe actions that would not be allowed without that combination.* Although these three CWEs differ for the underlying error, they all concern authorization problems and can thus be detected by verifying that a URL that cannot be reached by a user while navigating the user interface should not be available to that same user even when she directly requests the URL to the server, which is what drives OTG_AUTHZ_002 (i.e., an input sequence that is valid for a given user, should not lead to the same output when it is executed by another user, if it includes access to a URL with these characteristics).

Detailed information about the vulnerability types discovered by our extended catalog of MRs is provided in Appendix A.

Table 6: Catalog of MRs covering CWE Security Tactics View

MR ID
CWE_15_639_OTG_AUTHZ_004
CWE_22
CWE_73_99_219_220_530_OTG_AUTHZ_001a
CWE_79a_XSSreflected
CWE_79b_XSSreflected
CWE_79c_XSSstored
CWE_88_CommandInjection
CWE_88b_CommandInjection
CWE_89_943a_SQLinjection
CWE_89_943b_SQLinjection
CWE_90a_LDAPInjection
CWE_90b_LDAPInjection
CWE_93a_CRLF
CWE_93b_CRLF
CWE_94_96_StaticCodeInjection
CWE_923_RemoveCertificate
CWE_94_95_CodeInjection
CWE_138_150_OTG_AUTHZ_001b
CWE_258
CWE_262_263_309_324
CWE_266_267_268_OTG_AUTHZ_002
CWE_276_277_OTG_AUTHZ_002d
CWE_280_755_OTG_AUTHZ_002e
CWE_284_OTG_AUTHN_004
CWE_286_OTG_AUTHZ_002c
CWE_287a_425_OTG_AUTHN_001
CWE_288_287b_300_319_OTG_AUTHN_010
CWE_289a
CWE_289b_647
CWE_289b2
CWE_290_291
CWE_290_350
CWE_302
CWE_306_OTG_AUTHZ_002b
CWE_314
CWE_315
CWE_352
CWE_359_313_532_538_Log
CWE_420_OTG_CONFIG_007
CWE_434_Upload
CWE_471_472
CWE_488
CWE_521_WeakPassword
CWE_599_OutdatedCertificate
CWE_601_OTG_AUTHZ_002a
CWE_610_384
CWE_611_XMLInjectedFile
CWE_613_OTG_SESS_006
CWE_643_652a_XQueryInjection
CWE_643_652b_XQueryInjection
CWE_703_166_77_76_75_74_91_SpecialCharacters
CWE_757_OTG_CRYPST_004
CWE_784
CWE_792_793_794_795_796_797_SpecialCharacters
CWE_841

Table 7: Summary of the CWE architectural security design principles and weaknesses addressed by MST.

Security Design Principle	weaknesses		addressed weaknesses	
	all	generic	all	generic
Audit	6	6	1 (16%)	1 (16%)
Authenticate Actors	28	20	13 (46%)	13 (65%)
Authorize Actors	60	47	32 (53%)	31 (66%)
Cross Cutting	9	8	3 (33%)	3 (37%)
Encrypt Data	38	22	8 (21%)	8 (36%)
Identify Actors	12	8	3 (25%)	3 (37%)
Limit Access	8	7	3 (38%)	3 (43%)
Limit Exposure	6	4	0 (0%)	0 (0%)
Lock Computer	1	1	0 (0%)	0 (0%)
Manage User Sessions	6	3	4 (67%)	3 (100%)
Validate Inputs	39	33	31 (79%)	29 (88%)
Verify Message Integrity	10	10	2 (20%)	2 (20%)
Total	223	169	100 (45%)	96 (57%)

Table 8: Summary of the CWE Top 25 weaknesses addressed by MST.

Weaknesses		Addressed weaknesses	
all	generic	all	generic
25	17	13 (52%)	11 (64%)

Table 7 presents a summary of the CWE security design principles and related security weaknesses addressed by MST. The first column in Table 7 lists the security design principles appearing in the common security architectural tactics view. The second and third columns give, for each design principle, the overall number of weaknesses and the number of generic weaknesses, respectively. The fourth and fifth columns report the number (and percentage) of weaknesses and generic weaknesses that can be automatically discovered by MST, respectively.

In total, 100 out of all 223 weaknesses (45%) and 96 out of 169 generic weaknesses (57%) in the view can be addressed by MST. These numbers show that our approach enables engineers to automatically discover a large subset of the weaknesses.

MST can automatically discover a high percentage of generic weaknesses (above 60%) related to security design principles *Authenticate Actors*, *Authorize Actors*, *Manage User Sessions*, and *Validate Inputs* (i.e., 65%, 67%, 100%, and 88%, respectively). These weaknesses are about external systems or actors with invalid certification trying to access the system (*Authenticate Actors* and *Authorize Actors*), resources accessed by malicious users because of session management faults (*Manage User Sessions*), or providing malformed input data (e.g., code injection) to the system (*Validate Inputs*). On the other hand, MST addresses a low percentage of the generic weaknesses (below 20%) related to security design principles *Audit*, *Limit Exposure*, and *Lock*

Table 9: Summary of the security weaknesses for OWASP Top 10 security risks addressed by MST.

OWASP Security Risk	Weaknesses		Addressed weaknesses	
	all	generic	all	generic
Broken Access Control	20	19	16 (80%)	16 (84%)
Cryptographic Failures	23	22	4 (17%)	4 (18%)
Injection	23	19	18 (78%)	16 (84%)
Insecure Design	22	18	13(60%)	12 (67%)
Security Misconfiguration	5	4	3 (60%)	2 (50%)
Vulnerable and Outdated Component	0	0	0 (0%)	0 (0%)
Identification and Authentication Failures	20	20	12 (60%)	12 (60%)
Software and Data Integrity Failures	9	8	2 (22%)	2 (25%)
Security Logging and Monitoring Failures	4	4	1 (25%)	1 (25%)
Server-Side Request Forgery (SSRF)	1	1	0 (0%)	0 (0%)
Total	127	115	69 (54%)	65 (57%)

Table 10: Reasons preventing the application of MST.

ID	Reason
R0	The weakness cannot be discovered by means of user-system interactions.
R1	The weakness concerns a system that is not Web-based or mobile-based.
R2	The weakness can be discovered only by means of program analysis.
R3	It is not possible to distinguish valid and invalid behaviour based on system output; a human needs to inspect it.
R4	The weakness can be discovered only by means of data analysis.
R5	MST is inefficient. The weakness can be discovered only one test case execution.
R6	The weakness can be discovered only with results retrieved from a third side.

Table 11: Distribution of reasons preventing the application of MST to verify security design principles.

Security Design Principle	Weaknesses not addressed	R0	R1	R2	R3	R4	R5	R6	Sum
Audit	5			1		**2**	2		5
Authenticate Actor	15	2		7	1		**4**	1	15
Authorize Actor	28		**16**	6	2	2	2		28
Cross Cutting	6	1		3	2				6
Encrypt Data	30	**12**		**18**					36
Identify Actors	9	6	2				1		9
Limit Access	5		1		2		1	1	5
Limit Exposure	6			4	**6**				10
Lock Computer	1						1		1
Manage User Sessions	2		1		1				2
Validate Inputs	8		1	5				**2**	8
Verify Message Integrity	8			7			1		8
Total	133	21	21	**52**	15	4	12	4	133

Computer (i.e., 16%, 0%, and 0% respectively). MST relies on MRs that take as source inputs sequences of user-system interactions collected by a Web crawler. The weaknesses related to *Audit*, *Limit Exposure*, and *Lock Computer* are, on the contrary, about quality of recorded logs, information that the system exposes, and restrictions of the lockout mechanism (e.g., lock an account after a predefined number of failed logins). They all require manual data inspection.

The three security design principles associated with the highest number of weaknesses are *Authorize Actors* (32 weaknesses), *Validate Inputs* (31 weaknesses), and *Authenticate Actors* (13 weaknesses). Unsurprisingly, these design principles concern interactions between external actors and the system, which is the main focus of MST.

Table 8 gives a summary of the CWE Top 25 weaknesses addressed by our approach. MST can automatically discover 13 out of the 25 top weaknesses (52%) and 11 out of the 17 generic top weaknesses (64%), which shows that MST is a key solution to identify widely spread weaknesses.

Among the CWE Top 25 weaknesses, MST cannot address the ones that require program analysis or interactions with third parties (e.g., other system users or system administrators) to be detected: *Improper Restriction of Operations within the Bounds of a Memory Buffer*, *Cross-site Scripting*, *Information Exposure*, *Out-of-bounds Read*, *Use After Free*, *Integer Overflow or Wraparound*, *Cross-Site Request Forgery (CSRF)*, *Out-of-bounds Write*, *NULL Pointer Dereference*, *Improper Restriction of XML External Entity Reference*, *Improper Control of Generation of Code ('Code Injection')*, *Use of Hard-coded Credentials*, *Uncontrolled Resource*

Table 12: Distribution of reasons preventing the application of MST to discover weaknesses associated with the OWASP Top 10 security risks.

OWASP Security Risk	Weaknesses not addressed	R0	R1	R2	R3	R4	R5	R6	Sum
Broken Access Control	4	1	1	1				1	4
Cryptographic Failures	19	2		14	2	1			19
Injection	5			1	2			2	5
Insecure Design	9	1	1	4	1			1	9
Security Misconfiguration	2			2					3
Vulnerable and Outdated Component	0								0
Identification and Authentication	8		1	4	1			2	8
Software and Data Integrity Failures	7		1	3	3				7
Security Logging and Monitoring	3			3					3
Server-Side Request Forgery (SSRF)	1							1	1
Total	59	4	5	32	9	1	0	8	59

Consumption, *Missing Release of Resource after Effective Lifetime*, *Untrusted Search Path*, and *Deserialization of Untrusted Data*.

Table 9 presents a summary of the security weaknesses related to the OWASP Top 10 security risks addressed by our approach. MST can address 69 out of the 127 weaknesses (54%) and 65 out of the 115 generic weaknesses (57%) in this view. It addresses a high percentage (above 60%) of the weaknesses leading to security risks *Broken Access Control*, *Injection*, *Insecure Design*, *Security Misconfiguration*, and *Identification and Authentication Failures* (i.e., 80%, 78%, 60%, 60%, and 60%, respectively). These risks are about unauthorized access to resources, injecting malicious client-side scripts into a website, leveraging the lack of security controls (e.g., an unprotected primary channel), gaining system information thanks to system misconfiguration, and bypassing authentication. All involve malicious user-system interactions.

MST can address none of the weaknesses related to *Vulnerable and Outdated Components*, which concern the use of outdated libraries affected by known vulnerabilities. Although it would be feasible to implement MRs that detect failures of specific components, it is not possible for us to provide a catalog of MRs that cover all the outdated libraries in the market. Therefore we excluded them from the counting.

Table 10 presents the reasons preventing the application of MST to discover weaknesses. MST has been designed to generate source inputs by using either a Web crawler or manually implemented test scripts that automate the user-system interactions. Source inputs are automatically turned into follow-up inputs which are used to discover vulnerabilities. MST can discover vulnerabilities that can be exercised through a sequence of interactions. Therefore, MST cannot be applied when the attack is not based on user-system interactions (see R0 in Table 10) or when the weakness concerns a system that is not Web-based (see R1 in Table 10). Also, some weaknesses can be discovered only using program analysis, which MST does not support (see R2 in Table 10). In that case, the analysis should be either performed without actually executing programs (e.g., through code inspection) or the output of program analysis should be reviewed manually to eliminate false positives, which typically come in large numbers.

In some cases, it is not possible to define an MR because a human is needed to inspect the system output (R3). For instance, to discover weakness *Weak Password Recovery Mechanism for Forgotten Password* in Table 5, a human needs to indicate that the system under test contains a mechanism for the recovery of passwords that is weak (e.g., it is based on a security question whose answer can be easily determined [4]).

Due to MT fundamentals (based on MRs), MST cannot be applied for weaknesses that can be discovered only through data analysis (see R4 in Table 10). More precisely, some weaknesses can be determined only by analyzing large amounts of data (e.g., log files) based on statistics or machine learning; this analysis cannot be performed with an MR. Besides, it is not efficient to apply MT for weaknesses that can be discovered through a single test case execution (see R5 in Table 10). With a single test case execution, we do not need to generate follow-up inputs from a large number of source inputs. Finally, some weaknesses can be discovered only with results retrieved from a third side (e.g., passwords stored on a third-party server) or with interactions from a

Table 13: Distribution of reasons preventing the application of MST to discover CWE Top 25 weaknesses.

Weakness	R0	R1	R2	R3	R4	R5	R6
Out-of-bounds Write			1				
Out-of-bounds Read			1				
Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')		1					
Use After Free			1				
Integer Overflow or Wraparound				1			
Deserialization of Untrusted Data			1				
NULL Pointer Dereference			1				
Use of Hard-coded Credentials			1				
Improper Restriction of Operations within the Bounds of a Memory Buffer				1			
Incorrect Default Permissions	1						
Exposure of Sensitive Information to an Unauthorized Actor			1				
Server-Side Request Forgery (SSRF)							1
Total	1	1	**7**	2	0	0	1

third side (e.g., a victim clicks on a harmful URL provided by an attacker) (see R6 in Table 10). Expressing such interactions in MRs, in general, is infeasible.

Please note that R0, R2, and R4 represent three distinct cases. Indeed, R2 concerns weaknesses that require the system under test to be in a vulnerable state that is not feasible to reach with metamorphic testing but can be proved to be exploitable by an attacker thanks to static analysis approaches. R0 concerns weaknesses that can be discovered only through system execution using a complex input difficult to derive with a predefined transformation function used in an MR. R4 is associated with weaknesses that cannot be detected by verifying a single output but require the (statistical) analysis of multiple outputs.

Table 11 presents the distribution of reasons preventing the application of MST for the weaknesses regarding common security architectural tactics. Please note that there is more than one reason for some of the weaknesses (e.g., weakness *Small Space of Random Values* in Table 5). In 52 out of 133 weaknesses (39%) that cannot be discovered by MST, program analysis is required (see R2). This is expected since program analysis complements software testing in software verification.

The security design principle with most of the weaknesses not being addressed by MST is *Encrypt Data*, with 30 weaknesses not being addressed. 12 out of 30 weaknesses (40%) are not addressed due to R0. 18 out of 30 weaknesses (63%) are not addressed due to R2; these 18 weaknesses are associated with the protection of credentials or password (e.g., hard-coded cryptography key, password in configuration file), or the use of encryption/hash algorithm (e.g., hash without a salt). Therefore, in these cases, a static program analysis or code inspection approach should be used; for example, to find a hard-coded cryptography keys.

The second design principle with most of the weaknesses not being addressed by MST is *Authorize Actors*, for which MST cannot address 28 weaknesses. 16 out of these 28 weaknesses (57%) concern a system that is not Web-based (see R1). Indeed, these weaknesses affect the file system (e.g., preservation of permissions related to files, ownership management), the process control in an operation system (e.g., Linux), or the process communication in a mobile operation system (e.g., Android). Part of these weaknesses (e.g., the ones concerning components relevant to CPS development) will be covered by future developments in COSMOS.

Security design principles *Encrypt Data* and *Authorize Actors* include the largest number of weaknesses not being addressed because of reasons R0, R1, and R2. R3 affects mostly security design principle *Limit Exposure*, which indicates that the system should not provide information about its internal architecture or configuration (e.g., in error messages). To determine if this is an issue, it is necessary to inspect the system output. However, an MR cannot automatically capture (e.g., extract information from error messages) internal information of the system under test. A human is usually needed to inspect the system output.

Reason R4 concerns weaknesses mostly related to security design principle *Audit*; these weaknesses require to assess the quality of system logs (e.g., containing sensitive information, or containing too little information for incident analysis). In this regard, it is often necessary to rely on data analysis techniques using statistics or

machine learning to analyze system logs. Reason R5 concerns mostly weaknesses related to security design principle *Authenticate Actor*. Indeed, weaknesses related to design principle *Authenticate Actor* are about password management (e.g., weak password requirements, not using password aging) and can be detected through a single test case execution. Reason R6 concerns security design principle *Validate Inputs*. In some of these cases, to determine if the attack can be performed, we need interactions from a third actor (e.g., an administrator, a valid user), which we can perform with MRs only in certain cases.

Tables 12 and 13 report the reasons preventing the application of MST to discover some highly critical vulnerabilities. In these cases as well, the main reason is the necessity to rely on static program analysis. This is expected since testing and program analysis are complementary to quality assurance activities.

Summary. As a result of our analysis, we conclude that MST can address a large percentage (45%) of the weaknesses organized in the CWE view for common security architectural tactics. Such percentage grows up to 64% if we consider generic weaknesses only. MST can also address most high-risk weaknesses (54% of the weaknesses related to the OWASP Top 10 security risks and 52% of the CWE Top 25 weaknesses). These results are promising as they demonstrate that MST is relevant for a large subset of vulnerabilities occurring in practice. To our knowledge, among all the available research tools, MST is the one that automatically detects (i.e., it generates inputs and includes automated oracles) the largest set of vulnerability types. The weaknesses that MST cannot address are mostly those (i) that can be discovered only using program analysis, (ii) that are not based on user-system interactions, or (iii) that concern a non-web-based system.

4.3 MST for Automotive

We have evaluated the feasibility of relying on MST to detect security vulnerabilities for other relevant CPS cases. We considered the automotive context because (1) it concerns one COSMOS use case provider (i.e., AICAS) and (2) it is possible to find publicly available vulnerability reports for automotive systems. Since, differently from the context of Web-based systems, the CWE database does not provide vulnerability types specific to automotive, we have inspected the CVE database (see Section 2.1) looking for vulnerabilities affecting automotive components. Our objective is to rely on the information about each reported automotive CVE to derive MRs.

For our preliminary investigation, we have focused on vulnerabilities affecting the CAN bus - a widely used communication protocol in the automotive context. The CAN bus is a message-based protocol standard designed to allow microcontrollers and devices to communicate with each other; it is specified by the standard ISO-11898 [74]. Below, we report the textual description of the CVE records concerning the CAN bus as they appear in the CVE Web site[17]:

- *CVE-2015-5611* “Unspecified vulnerability in Uconnect before 15.26.1, as used in certain Fiat Chrysler Automobiles (FCA) from 2013 to 2015 models, allows remote attackers in the same cellular network to control vehicle movement, cause human harm or physical damage, or modify dashboard settings via vectors related to modification of entertainment-system firmware and access of the CAN bus due to insufficient "Radio security protection," as demonstrated on a 2014 Jeep Cherokee Limited FWD”.
- *CVE-2016-9337* “An issue was discovered in Tesla Motors Model S automobile, all firmware versions before version 7.1 (2.36.31) with web browser functionality enabled. The vehicle’s Gateway ECU is susceptible to commands that may allow an attacker to install malicious software allowing the attacker to send messages to the vehicle’s CAN bus, a Command Injection”.
- *CVE-2017-14937* The airbag detonation algorithm allows injury to passenger-car occupants via predictable Security Access (SA) data to the internal CAN bus (or the OBD connector). This affects the airbag control units (aka pyrotechnical control units or PCUs) of unspecified passenger vehicles manufactured in 2014 or later, when the ignition is on and the speed is less than 6 km/h. Specifically, there are only 256 possible key pairs, and authentication attempts have no rate limit. In addition, at least one

[17]For our search, we relied on the google.com search engine with the search string “*"CAN bus" site:cve.mitre.org*”.

manufacturer's interpretation of the ISO 26021 standard is that it must be possible to calculate the key directly (i.e., the other 255 key pairs must not be used). Exploitation would typically involve an attacker who has already gained access to the CAN bus, and sends a crafted Unified Diagnostic Service (UDS) message to detonate the pyrotechnical charges, resulting in the same passenger-injury risks as in any airbag deployment.

- *CVE-2020-8539* "Kia Motors Head Unit with Software version: SOP.003.30.18.0703, SOP.005.7.181019, and SOP.007.1.191209 may allow an attacker to inject unauthorized commands, by executing the micomd executable deamon, to trigger unintended functionalities. In addition, this executable may be used by an attacker to inject commands to generate CAN frames that are sent into the M-CAN bus (Multimedia CAN bus) of the vehicle".
- *CVE-2020-29440* "Tesla Model X vehicles before 2020-11-23 do not perform certificate validation during an attempt to pair a new key fob with the body control module (BCM). This allows an attacker (who is inside a vehicle, or is otherwise able to send data over the CAN bus) to start and drive the vehicle with a spoofed key fob".

Unfortunately, automotive vulnerabilities are difficult to exploit and test (a precondition to understand them), mainly because the vulnerable hardware (i.e., a car) is required. To overcome such limitation and be able to study such vulnerabilities, one possible solution consists of relying on publicly available exploit scripts[18] and hardware simulators (e.g., simulators for ECU). However, only two of the vulnerabilities listed above are accompained by a detailed explanation; they are CVE-2020-8539[19] and CVE-2017-14937[20], which has been detailed in a research paper [56]. However, only for CVE-2017-14937 an exploit is available[21].

For the reason above, we focused on CVE-2017-14937. However, unfortunately, the available ECU simulators (see Section 2.1) do not implement the features required to successfully execute the exploit for CVE-2017-14937; therefore, we extended the Duraki's VirtualCar simulator (see Section 2.1) to include all the features required to make the exploit script succeed (i.e., appropriately respond to the message requests made by the exploit script). Our extension to the Duraki's VirtualCar simulator enabled us to have a clear understanding of the vulnerability [22].

Below, we describe a MR derived from CVE-2017-14937 and our extensions to the MST framework to support the implementation of the MR.

The vulnerability CVE-2017-14937 concerns the Unified Diagnostic Services (UDS) communication protocol, which is a diagnostic communication protocol used in electronic control units (ECUs) within automotive electronics; UDS is specified in the standard ISO 14229-1 [76]. UDS is adopted by suppliers of Original Equipment Manufacturer (OEM). The ECUs in modern vehicles control a wide range of functions, including electronic fuel injection, engine control, transmission, anti-lock braking system, door locks, braking.

UDS communication occurs through the CAN bus. The CAN protocol specifies the first and second layer of the OSI Model - that is the Physical Layer and the Data Link Layer. UDS, however, also specifies the fifth (Session Layer) and seventh (Application Layer) layers of the OSI Model. The Service ID (SID) and the parameters associated with the services are contained in the payload of a message frame.

Within UDS, a Security Access Request message is used to enable the most security-critical services; it is sent from the client to the control unit. A *seed* is then generated and sent to the client by the control unit within a Security Access Response message; from this *seed*, the client computes a *key* and sends it back to the control unit to unlock the security-critical services.

[18] We use the term exploit script (or simply *exploit*) to indicate an executable program that sends a sequence of commands to the software under test to successfully exercise the vulnerablity and violate the software security properties.

[19] An detailed explanation is available at https://sowhat.iit.cnr.it/pdf/IIT-20-2020.pdf

[20] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-14937

[21] See `https://github.com/rapid7/metasploit-framework/blob/master/modules/post/hardware/automotive/pdt.rb`

[22] Our extension is available at `https://gitlab.uni.lu/cosmos/durakivirtualcar`

However, CVE-2017-14937 reports that at least one manufacturer's interpretation of the ISO 26021 standard is that it must be possible to calculate the key directly (i.e., the key never varies).

To implement a MR that detects CVE-2017-14937 and similar problems (i.e., ECUs using an hardcoded key), MST shall provide features that enable dealing with message exchanges through the CAN bus. To this end, we have extended the MST framework with the following concepts:

- *CANMessageSequence* class. It represents a sequence of messages exchanged over the can bus. Such message sequences can be recorded through the *candump* tool [95], which is a utility that connects to a Linux virtual CAN bus and visualizes all the messages being exchanged [23]. Figure 27, for example, shows the sequence of messages exchanged to exploit the vulnerability CVE-2017-14937 using the Metasploit framework (Figure 28). A CANMessageSequence can be either a source input or a follow-up input. In our context, source inputs consist of the sequence of messages exchanged during functional testing; precisely, we consider the messages sent by a testing framework to the control unit under test (e.g., the messages sent by the device with ID 7F1, in Figure 27).
- *CANMessage* class. It represents a single message belonging to a CANMessageSequence.
- *CANMessageSequence(int)*, which is a data function. It is the function used to refer to CANMessage-Sequence used as either source or follow-up input. It corresponds to the *Input(int)* data function used by MST in the Web context to identify source and follow-up inputs.
- *isSecurityAccessRequest(CANMessage m)*, which is a CAN-specific function. It is a utility function used to determine if a CANMessage is a Security Access Request, that is, if it starts with the service ID 0x27.
- *CANOutput(CANMessageSequence s)*, which is a CAN-specific function. It is a utility function used to return the sequence of messages sent in response to a CANMessageSequence.

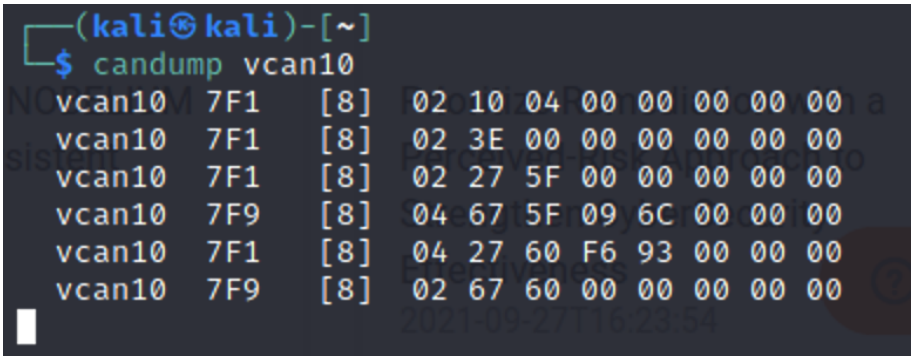


Figure 27: Output of candump when exploiting CVE-2017-14937.

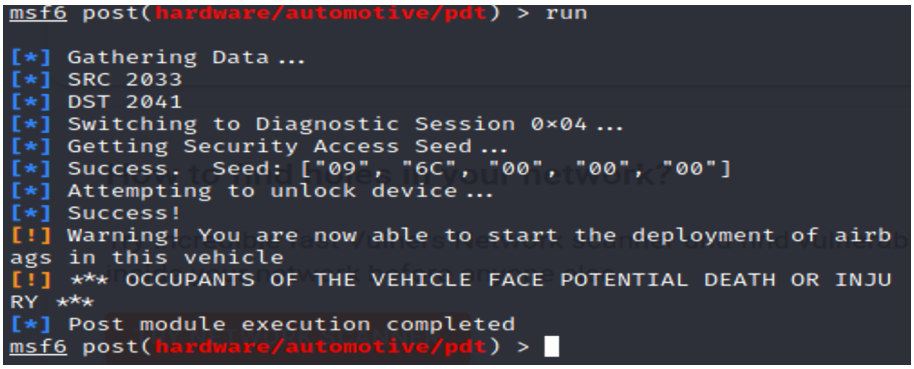


Figure 28: Output of Metasploit when exploiting CVE-2017-14937.

[23] The virtual CAN bus can be connected to a real CAN device.

```
import static smrl.mr.language.Operations.*;
import static smrl.mr.automotive.CANOperations.*;
import smrl.mr.automotive.CANMessage

package smrl.mr.automotive {

/**
 * This MR is inspired by 'CVE-2017-14937'
 *
 * Within the Unified Diagnostic Services (UDS) communication protocol,
 * a Security Access message is used to enable the most security-critical services.
 *
 * For this purpose a "Seed" is generated and sent to the client by the control unit.
 * From this "Seed" the client has to compute a "Key" and send it back
 * to the control unit to unlock the security-critical services.
 *
 *
 * However, CVE-2017-14937 reports that at least one manufacturer's interpretation of
 * the ISO 26021 standard is that it must be possible to calculate the key directly
 * (i.e., the other 255 key pairs must not be used).
 *
 * This MR simply creates a follow-up input that is a replica of a source input message sequence.
 * If the system is not vulnerable, it shall generate a seed that is different for the two input sequences.
 * Consequently, the second sequence of CAN messages shall prevent enabling security-critical services;
 * therefore, it shall lead to an output that is different from the output generated by the source input.
 *
 */
MR UDS_PredictableSecurityAccess {
    {
            for ( CANMessage m : CANMessageSequence(1).messages() ){
                IMPLIES (
                    isSecurityAccessRequest(m) &&
                    EQUAL ( CANMessageSequence(2), CANMessageSequence(1) )
                    ,
                    ! CANOutput( CANMessageSequence(2) ).equals ( CANMessageSequence(1) )
                )
            }
    }
}
}
```

Figure 29: MR UDS_PredictableSecurityAccess

Based on CVE-2017-14937, we have defined a MR called *UDS_PredictableSecurityAccess* that creates a follow-up input that is a replica of a source input message sequence. It is shown in Figure 29. Our MR relies on the key idea that if the control unit is not vulnerable (i.e., the key varies), it shall generate a seed that is different for the two input sequences. Consequently, the second sequence of CAN messages shall prevent enabling security-critical services. Therefore, the output sequence generated by the control unit for the follow-up input shall be different from the one generated for the source input (i.e., access shall be prevented). Please note that our MR detects security issues not detected by the Metasploit exploit; indeed, the Metasploit exploit determines if the key is generated as the ones' complement of the seed, we deal with the case of having a key with a fixed value.

Although not extensive, our study is a first step towards demonstrating the feasibility of implementing automotive MRs and applying MST in the automotive context. Our future work includes extending our set of MRs to additional subjects in the automotive and other CPS contexts.

5 Misuse Case Programming: Extended MCP toolset

In this section we describe the extensions provided to the MCP toolset [99, 100], which concerns handling misuse case specifications written for programs different than Web systems and providing new NLP-based solutions that analyze complex sentences containing infinitive phrases.

5.1 Restricted Misuse Case Modelling

Table 14 provides two examples of misuse case specifications written according to the Restricted Misuse Case Modelling approach (RMCM), which are referred to in this Section.

Table 14: Example misuse case specifications written for a system in the medical domain.

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.
34	
35	**MISUSE CASE** Record Forged Weight
36	**Description** The MALICIOUS user creates a specific record for the Weight Activity of another user.
37	**Precondition** The MALICIOUS user is logged into the system, he must know the user id associated to another user.
38	**Basic Threat Flow**
39	1. The MALICIOUS user configures the proxy to replace his user id with the VICTIM user id.
40	2. The MALICIOUS user provides an anomalous weight to the system.
41	3. The MALICIOUS user refreshes the App.
42	4. The VICTIM's CARER provides his username and his password to the login page of the Web interface.
43	5. The VICTIM's CARER retrieves the weight page of the VICTIM from the Web interface.
44	6. The MALICIOUS user has exploited the system.
45	**Postcondition:** The MALICIOUS user has generated a record for another user.
46	**Specific Alternative Flow (SAF1)**
47	RFS 6
48	1. IF The anomalous weight is not equal to the weight value of the Web interface THEN
49	2. ABORT
50	3. ENDIF
51	**Postcondition:** The MALICIOUS user did not forge the weight record.

As indicated in Section 2.3.1, RMCM relies on the keywords `SENDS ... TO` and `REQUESTS ... FROM` in the form of `<ActorA> SENDS <something> TO <ActorB>` and `<ActorA> REQUESTS <something> FROM <ActorB>` for the system-actor interactions, with `<ActorA>` and `<ActorB>` being either the name of an actor or the keyword *the system*, which indicates the system under test. Example sentences are *The MALICIOUS user requests the resource from the system* (Line 14 in Table 14) and *The MALICIOUS user sends username and password to the system through the login page* (Line 14). Although these two keywords are sufficient for writing misuse case specifications for Web systems, they are inappropriate to express misuse case specifications for applications that are not executed on a remote server (e.g., a desktop terminal or a tablet App). Indeed, end-users of such applications do not have the impression of sending data to the application under test while interacting with it. For this reason, in the updated version of the MCP toolset, we do not restrict the set of verbs to be used for system-actor interactions. For inputs and outputs, any verb characterizing the act of transferring something to another actor can be used; for example, the verb *to provide* in the sentence *The MALICIOUS user provides an anomalous weight to the system* (Line 14). Also, we allow the description of system-actor interactions where the system under test is not mentioned explicitly (e.g., *The MALICIOUS user enters the leaderboard area*).

5.2 Natural Language Processing

Natural language processing (NLP) refers to techniques that aim to process text written in natural language to extract information [79, 80]. Several NLP techniques exist; they differ for underlying algorithms and objectives. Part-of-speech (POS) tagging approaches assign words with tags capturing their syntactic categories (e.g., noun, verb, pronoun, preposition, adverb, conjunction, participle, and article). Syntactic parsing techniques assign a syntactic structure to sentences. They identify constituents, i.e., groups of words behaving as a single unit. Please note that many constituents are phrases. Syntactic parsing techniques differ for the underlying formalisms, e.g., Cocke-Kasami-Younger (CKY) algorithm [82, 154], Probabilistic Context-Free Grammars (PCFGs) [85], or dependency grammars [53].

One extension that we provided to the MCP toolset is the capability to determine infinitive phrases; to this end we employ the Stanford constituency parser, which relies on a PCFG [85]. Fig. 30 presents the output generated by the Stanford constituency parser for the sentence in Line 14 in Table 14 (i.e., "*The MALICIOUS user configures the proxy to replace his user id with the VICTIM user id*"). In Fig. 30, we report constituents in capital letters. Constituent names are defined by the Penn Treebank project [104]. The sentence in Fig. 30 contains the infinitive phrase "*to replace his user id with the VICTIM user id*"; consequently, the Stanford constituency parser indicates the presence of two nested sentences (see the nested constituent of type *S* in Line 8 in Fig. 30).

Another extension to the MCP toolset is the capability to determine possessive forms in a sentence; to this end, we rely on the Stanford dependency parser [42], which analyzes the grammatical structure. It establishes relationships between a pair of words by specifying how a *dependent* word modifies a *governor* word. The Stanford dependency parser provides labels for 50 different grammatical relations [52]. MCP relies on the relation *possessive*, which holds between the head of an NP and the genitive *'s*, and the relation *poss*, which holds between the head of a noun phrase and its possessive determiner, or a genitive *'s* complement. Fig. 31 shows the output of the Stanford dependency parser for the sentence "*The VICTIM's CARER provides his username and his password to the login page of the Web interface*". The arrows show dependency relationships annotated with labels describing the nature of the dependency. A *poss* relation holds between *CARER* and *VICTIM* (it indicates that we are referring to the CARER of a VICTIM for the attack). A *possessive* relation holds between *VICTIM* and the genitive *'s*.

The updated MCP toolset still relies on Semantic Role Labeling (SRL) like the previous version. SRL techniques are capable of automatically determining the roles played by words in a sentence. For the sentences "*The system starts*" and "*The system starts the database*", SRL can determine that the actors affected by the actions are "*the system*" and "*the database*", respectively. The component that is started coincides with the

```
1. (ROOT
2.  (S (NP (DT The)
3.     (NNP MALICIOUS)
4.     (NN user))
5.     (VP (VBZ configures)
6.       (NP (DT the)
7.           (NN proxy)
8.           (S (VP (TO to)
9.     (VP (VB replace)
10.              (NP (PRP$ his)
11.                  (NN user)
12.                  (NN id))
13.             (PP (IN with)
14.                  (NP (DT the)
15.                      (NNP VICTIM)
16.                      (NN user)
17.                      (NN id))))))))
18.   (. .)))
```

Legend: DT: Determiner; IN: Preposition or subordinating conjunction; NN: Noun, singular or mass; NNP: Proper noun, singular NP: Noun phrase; PP: Prepositional phrase; PRP$ Possessive pronoun; S: sentence; TO to; VP: Verb phrase; VBZ: Verb, 3rd person singular present.

Figure 30: Example output generated by the Stanford constituency parser for the sentence in Line 14 of Table 14.

Table 15: PropBank Additional Semantic Roles used in the paper.

Verb-specific semantic roles	
Identifier	**Definition**
A0	Usually indicates who performs an action.
A1	Usually indicates the actor most directly affected by the action.
A2	With motion verbs, indicates a final state or a location.
Generic semantic roles	
Identifier	**Definition**
AM-ADV	Adverbial modification.
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-TMP	Provides temporal information.
AM-PRD	Secondary predicate with additional information about A1.

subject in the first sentence and with the object in the second sentence, although the verb *to start* is used with the active voice in both. This information cannot be captured by other NLP techniques like POS tagging or dependency parsing.

There are few SRL tools [37, 148, 147]. MCP relies on the CogComp NLP pipeline (hereafter CNP [147]), which uses the PropBank [121] and NomBank models [110, 64]. PropBank models enable tagging the words in a sentence with keywords (e.g., *A0*, *A1*, *A2*, and *AN*) to indicate their roles. *A0* indicates who performs an action, while *A1* indicates the actor most directly affected by the action. For instance, the term "*The system*" is tagged with *A1* in the sentence "*The system starts*", while the term "*the database*" is tagged with *A1* in the sentence "*The system starts the database*". The other roles are verb-specific despite some commonalities, e.g., *A2* which is often used for the end state of an action.

PropBank includes additional roles that are not verb-specific (see Table 15). They are labeled with general keywords and match adjunct information in different sentences, e.g., *AM-NEG* indicating negative verbs. NomBank, instead, captures the roles of nouns, adverbs, and adjectives in noun phrases. It uses the same keywords adopted by PropBank. For instance, using ProbBank, we identify that the noun phrase *dictionary values* plays the role *A1* in the sentence "*The malicious user sends dictionary values*". Using NomBank, we obtain complementary information indicating the term *values* is the main noun (tagged with *A0*), and the term *dictionary* is an attributive noun (tagged with *A1*).

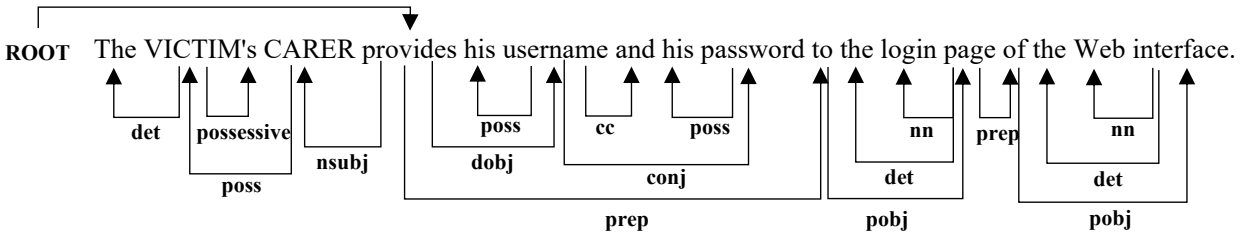


Figure 31: Dependency parsing example.

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

Figure 32: Example SRL tags generated by CNP.

Fig. 32 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. Figs. 33 and 34 provide the SRL output for the sentences in Figs. 30 and 31, respectively. Fig. 33 shows that, in the presence of multiple sentences, CNP enables the identification of separate SRL roles for the distinct sentences. Fig. 34 exemplifies how the SRL output is simpler to process than the output of the dependency parser. The elements affected by the verb (i.e., "*his username and password*") are explicitly identified in the same block of words by CNP. Instead, in the Stanford dependency parser, they can be identified only by traversing multiple relations (see Fig. 31). However, SRL provides coarse-grained information that may not be enough to match phrases to API names. For example, it does not enable MCP to identify possession relations (the phrase "*of the Web interface*" is part of the block of words tagged as *A2* in Fig. 34). Therefore, the MCP toolset combines the output of SRL and dependency parsing.

5.3 Identification of Test Inputs

MCP automatically identifies input entities, determines relationships between input entities (e.g., each *username* is associated to a *role*), and assists engineers in generating values to be assigned to input entities.

MCP assumes that input entities appear in misuse case steps with a verb that indicates one or more entities being provided 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 determine sentences having these entities. Depending on the verb, SRL usually tags entity destinations with roles `A2` or `AM-LOC` (see Section 5.2). Therefore, it is highly likely that a sentence having phrases tagged with `A2` or `AM-LOC` describes an input activity. In such a sentence, MCP looks for entities that match phrases tagged with `A1` (i.e., the ones affected by the verb).

We have extended MCP to handle infinitive sentences describing actions beyond input provision; for these cases, the updated MCP identifies input entities by considering every semantic role except *A0*, which stands for the actor performing the action. For example, when processing the sentence "*to replace his user id with the*

{The MALICIOUS user}$_{A0}$ {configures}$_{verb}$ {the proxy}$_{A1}$ to {replace}$_{verb}$ {his user id}$_{A1}$ with {the VICTIM user id}$_{A2}$.

Figure 33: Example SRL tags generated by CNP for the sentence in Fig. 30. We use gray for the infinitive sentence.

{The VICTIM's CARER}$_{A0}$ {provides}$_{verb}$ {his username and his password}$_{A1}$ {to the login page of the Web interface}$_{A2}$

Figure 34: Example SRL tags generated by CNP for the sentence in Figure 31.

ReplaceWeightRecord.java

```
24    public void testCase() throws IOException {
25        system =new SystemUnderTest();
26        AndroidTester maliciousUser = this;
27        webInterface=new WebInterface();
28        maliciousUser.replace(input.get("malicious_user_user_id"),input.get("victim_user_id"));
29        system.provideWeight(input.get("anomalous_weight"));
30        system.refreshApp();
31        Map<String,Object> parameters =new HashMap();
32        parameters.put("victim_carer_password",input.get("victim_carer_password"));
33        parameters.put("victim_carer_username",input.get("victim_carer_username"));
34        webInterface.provide(input.get("login_page_OF_web_interface"),parameters);
35        webInterface.retrieve(input.get("weight_page_OF_victim"));
36        if(input.get("anomalous_weight").equals(webInterface.weight)){
37            maliciousUser.exploit();
38            maliciousUser.exit("The MALICIOUS user has generated a record for another user");
39        }
40        else{
41            maliciousUser.abort("The MALICIOUS user did not generate a forged record");
42        }
```

Figure 35: Part of test case automatically generated for misuse case 'Record Forged Weight' in Table 14.

VICTIM user id" in Fig. 33, MCP analyzes input entities both *his user id* (tagged with A1) and *the VICTIM user id* (tagged with A2).

5.4 Generation of Executable Test Cases

MCP automatically generates an executable test case for each misuse case specification (Phase 4 in Fig. 11). The current MCP prototype supports test case generation both in Python and Java and follows the same process for both two languages. The only differences concern the selection of language-specific elements. For example, in Java, `Map` is used instead of `Dict`, and each generated test case is a JUnit test case. In Python, each test case is a Python class implementing method `run`. Figs. 14 (page 24) and 35 present the test cases generated from misuse case specifications *Bypass Authorization Schema* (Python) and *Record Forged Weight* (Java), respectively.

MCP declares and initializes three variables, `system`, `maliciousUser`, and `inputs` (Lines 3, 4, and 5 in Fig. 14). Variable `system` refers to an instance of class `System`, which provides methods that trigger the system functions under test (e.g., `request`). Variable `maliciousUser` refers to the instance of the test class simulating the behavior of the malicious user. Variable `inputs` refers to a dictionary populated with the input values specified in the JSON input file. The end-user can configure MCP to initialize additional variables for components or actors involved in the misuse cases. For example, in Line 27 in Fig. 35, variable `webInterface` has been initialized with a class instance exposing methods that access the Web interface of the system under test.

The updated MCP also processes sentences with infinitive phrases (e.g., the sentence "*The MALICIOUS user configures the proxy to replace his user id with the VICTIM user id.*" in Line 14, Table 14). It relies on the Stanford parser output to determine if a misuse case step contains a sentence with an infinitive phrase (see Fig. 30). In general, we expect that the sentence before the infinitive particle (e.g., "*The MALICIOUS user configures the proxy*") describes how the system or the malicious user performs an action. In other words, it should provide implementation details that are likely not captured by API method names. The infinitive phrase,

which expresses the action objective, should match the name of a test driver API method. To avoid imposing a writing rule, MCP generates two method calls: one call for part of the sentence before the infinitive particle (e.g., "*The MALICIOUS user configures the proxy*") and another for the infinitive phrase (e.g., "*to replace his user id with the VICTIM user id*"). It then selects the method call that has the highest similarity with the sentence in the specifications (see the original MCP paper for details [99]).

6 Conclusion

This report concerned *COSMOS Task T4.3: Development of Solutions for Detecting Security Vulnerabilities in CPS*; we reported about the period M1-M12, which concerned the definition of specification-based techniques for automated security testing.

Our developments mainly concerned MST, a technique that ensures that both documented and unknown (or partially documented) software misuses that break the security properties of the software are infeasible. It relies on a set of manually written specifications, which are called metamorphic relations (MRs). During the reported period, COSMOS has extended the available catalog of MRs thus enabling the identification of 100 vulnerability types concerning mistakes in the application of security design principles, which is a result that is clearly valuable for software developers. To enable the implementation of the MR catalog, we have also extended the functionalities of the MST toolset. In parallel, we extended MCP, a technique that enables the automated generation of test cases based on documented software misuses. In COSMOS, MCP will be used for the discovery of security vulnerabilities that cannot be discovered by MST.

In our work, we mainly focused on the development of approaches that work with Web systems because Web systems are often used to access and control CPSs. Future work concerns addressing CPS-specific interfaces. Also, we will report on an ongoing empirical evaluation of the effectiveness of the provided catalog of MRs. Finally, we will address challenges related to minimizing testing cost and automatically extending security vulnerability testing capabilities by automatically generating MRs through machine learning.

A CWEs covered by the Extended Catalog of Security MRs

In Table 16, we provide the list of CWEs appearing in the *common security architectural tactics* view of the CWE database [9], grouped by security concept. We also report an indication of the CWEs covered by our extended catalog of MRs.

Table 16: CWEs appearing in the *common security architectural tactics* view of the CWE database [9], grouped by security concept, and belonging to our extended catalog of MRs.

Design concept	Weakness (CWE ID)	Covered by the extended MST catalog of MRs
Audit		1
1.1	Improper Output Neutralization for Logs - (117)	NO
1.2	Omission of Security-relevant Information - (223)	NO
1.3	Obscured Security-relevant Information by Alternate Name - (224)	NO
1.4	Inclusion of Sensitive Information in Log Files - (532)	YES
1.5	Insufficient Logging - (778)	NO
1.6	Logging of Excessive Data - (779)	NO
Authenticate Actors		13
2.1	Empty Password in Configuration File - (258)	YES
2.2	Use of Hard-coded Password - (259)	NO
2.3	Not Using Password Aging - (262)	YES
2.4	Password Aging with Long Expiration - (263)	YES
2.5	Improper Authentication - (287)	YES
2.6	Authentication Bypass Using an Alternate Path or Channel - (288)	YES
2.7	Authentication Bypass by Alternate Name - (289)	YES
2.8	Authentication Bypass by Spoofing - (290)	YES
2.9	Reliance on IP Address for Authentication - (291)	YES
2.10	Using Referer Field for Authentication - (293)	NO
2.11	Authentication Bypass by Capture-replay - (294)	NO
2.12	Reflection Attack in an Authentication Protocol - (301)	NO
2.13	Authentication Bypass by Assumed-Immutable Data - (302)	YES
2.14	Incorrect Implementation of Authentication Algorithm - (303)	NO
2.15	Missing Critical Step in Authentication - (304)	YES
2.16	Authentication Bypass by Primary Weakness - (305)	YES
2.17	Missing Authentication for Critical Function - (306)	YES
2.18	Improper Restriction of Excessive Authentication Attempts - (307)	NO
2.19	Use of Single-factor Authentication - (308)	NO
2.20	Key Exchange without Entity Authentication - (322)	NO
2.21	Weak Password Requirements - (521)	YES
2.22	Authentication Bypass: OpenSSL CTX Object Modified after SSL Objects are Created - (593)	NO
2.23	Use of Client-Side Authentication - (603)	NO
2.24	Unverified Password Change - (620)	NO
2.25	Weak Password Recovery Mechanism for Forgotten Password - (640)	NO
2.26	Use of Hard-coded Credentials - (798)	NO
2.27	Use of Password Hash Instead of Password for Authentication - (836)	NO
2.28	Use of Password Hash With Insufficient Computational Effort - (916)	NO
Authorize Actors		32
3.1	Process Control - (114)	NO
3.2	External Control of System or Configuration Setting - (15)	YES
3.3	Sensitive Data Under Web Root - (219)	YES
3.4	Sensitive Data Under FTP Root - (220)	YES
3.5	Incorrect Privilege Assignment - (266)	YES
3.6	Privilege Defined With Unsafe Actions - (267)	YES
3.7	Privilege Chaining - (268)	YES
3.8	Improper Privilege Management - (269)	YES
3.9	Privilege Context Switching Error - (270)	NO
3.10	Privilege Dropping / Lowering Errors - (271)	NO
3.11	Least Privilege Violation - (272)	NO
3.12	Improper Check for Dropped Privileges - (273)	NO
3.13	Improper Handling of Insufficient Privileges - (274)	NO
3.14	Incorrect Default Permissions - (276)	YES
3.15	Insecure Inherited Permissions - (277)	YES
3.16	Incorrect Execution-Assigned Permissions - (279)	NO
3.17	Improper Handling of Insufficient Permissions or Privileges - (280)	YES
3.18	Improper Preservation of Permissions - (281)	NO
3.19	Improper Ownership Management - (282)	NO
3.20	Unverified Ownership - (283)	NO
3.21	Improper Access Control - (284)	YES
3.22	Improper Authorization - (285)	YES
3.23	Incorrect User Management - (286)	YES
3.24	Channel Accessible by Non-Endpoint ('Man-in-the-Middle') - (300)	YES
3.25	Predictable from Observable State - (341)	NO
3.26	Exposure of Private Information ('Privacy Violation') - (359)	YES
3.27	Exposure of File Descriptor to Unintended Control Sphere ('File Descriptor Leak') - (403)	NO
3.28	Unprotected Primary Channel - (419)	NO
3.29	Unprotected Alternate Channel - (420)	YES
3.30	Direct Request ('Forced Browsing') - (425)	NO
3.31	Untrusted Search Path - (426)	NO
3.32	Unrestricted Upload of File with Dangerous Type - (434)	YES
3.33	Exposure of CVS Repository to an Unauthorized Control Sphere - (527)	NO
3.34	Exposure of Core Dump File to an Unauthorized Control Sphere - (528)	YES
3.35	Exposure of Access Control List Files to an Unauthorized Control Sphere - (529)	YES
3.36	Exposure of Backup File to an Unauthorized Control Sphere - (530)	YES
3.37	File and Directory Information Exposure - (538)	NO
3.38	Incorrect Behavior Order: Authorization Before Parsing and Canonicalization - (551)	YES
3.39	Files or Directories Accessible to External Parties - (552)	YES
3.40	Authorization Bypass Through User-Controlled SQL Primary Key - (566)	NO
3.41	Authorization Bypass Through User-Controlled Key - (639)	YES
3.42	External Control of Critical State Data - (642)	YES
3.43	Use of Non-Canonical URL Paths for Authorization Decisions - (647)	YES
3.44	Insufficient Compartmentalization - (653)	NO
3.45	Reliance on Security Through Obscurity - (656)	NO
3.46	Exposure of Resource to Wrong Sphere - (668)	YES
3.47	Incorrect Resource Transfer Between Spheres - (669)	YES
3.48	Lack of Administrator Control over Security - (671)	NO
3.49	External Influence of Sphere Definition - (673)	NO
3.50	Incorrect Ownership Assignment - (708)	NO
3.51	Incorrect Permission Assignment for Critical Resource - (732)	YES
3.52	Allocation of Resources Without Limits or Throttling - (770)	NO
3.53	Exposed IOCTL with Insufficient Access Control - (782)	NO
3.54	Improper Control of Document Type Definition - (827)	NO
3.55	Missing Authorization - (862)	YES

3.56	Incorrect Authorization - (863)	YES
3.57	Storage of Sensitive Data in a Mechanism without Access Control - (921)	YES
3.58	Improper Restriction of Communication Channel to Intended Endpoints - (923)	YES
3.59	Improper Authorization in Handler for Custom URL Scheme - (939)	NO
3.60	Overly Permissive Cross-domain Whitelist - (942)	NO
Cross Cutting		3
4.1	Information Exposure Through Timing Discrepancy - (208)	NO
4.2	Missing Report of Error Condition - (392)	NO
4.3	Improper Cleanup on Thrown Exception - (460)	NO
4.4	Missing Standardized Error Handling Mechanism - (544)	NO
4.5	Client-Side Enforcement of Server-Side Security - (602)	NO
4.6	Improper Check or Handling of Exceptional Conditions - (703)	YES
4.7	Improper Check for Unusual or Exceptional Conditions - (754)	NO
4.8	Reliance on Cookies without Validation and Integrity Checking in a Security Decision - (784)	YES
4.9	Reliance on Untrusted Inputs in a Security Decision - (807)	YES
Encrypt Data		8
5.1	Unprotected Storage of Credentials - (256)	NO
5.2	Storing Passwords in a Recoverable Format - (257)	NO
5.3	Password in Configuration File - (260)	NO
5.4	Weak Cryptography for Passwords - (261)	NO
5.5	Missing Encryption of Sensitive Data - (311)	NO
5.6	Cleartext Storage of Sensitive Information - (312)	YES
5.7	Cleartext Storage in a File or on Disk - (313)	YES
5.8	Cleartext Storage in the Registry - (314)	YES
5.9	Cleartext Storage of Sensitive Information in a Cookie - (315)	YES
5.10	Cleartext Storage of Sensitive Information in Memory - (316)	NO
5.11	Cleartext Storage of Sensitive Information in GUI - (317)	NO
5.12	Cleartext Storage of Sensitive Information in Executable - (318)	NO
5.13	Cleartext Transmission of Sensitive Information - (319)	YES
5.14	Use of Hard-coded Cryptographic Key - (321)	NO
5.15	Reusing a Nonce, Key Pair in Encryption - (323)	NO
5.16	Use of a Key Past its Expiration Date - (324)	YES
5.17	Missing Required Cryptographic Step - (325)	NO
5.18	Inadequate Encryption Strength - (326)	NO
5.19	Use of a Broken or Risky Cryptographic Algorithm - (327)	NO
5.20	Reversible One-Way Hash - (328)	NO
5.21	Use of Insufficiently Random Values - (330)	NO
5.22	Insufficient Entropy - (331)	NO
5.23	Insufficient Entropy in PRNG - (332)	NO
5.24	Improper Handling of Insufficient Entropy in TRNG - (333)	NO
5.25	Small Space of Random Values - (334)	NO
5.26	Incorrect Usage of Seeds in Pseudo-Random Number Generator (PRNG) - (335)	NO
5.27	Same Seed in Pseudo-Random Number Generator (PRNG) - (336)	NO
5.28	Predictable Seed in Pseudo-Random Number Generator (PRNG) - (337)	NO
5.29	Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG) - (338)	NO
5.30	Small Seed Space in PRNG - (339)	NO
5.31	Improper Verification of Cryptographic Signature - (347)	NO
5.32	Insufficiently Protected Credentials - (522)	YES
5.33	Unprotected Transport of Credentials - (523)	NO
5.34	Selection of Less-Secure Algorithm During Negotiation ('Algorithm Downgrade') - (757)	YES
5.35	Use of a One-Way Hash without a Salt - (759)	NO
5.36	Use of a One-Way Hash with a Predictable Salt - (760)	NO
5.37	Use of RSA Algorithm without OAEP - (780)	NO
5.38	Insecure Storage of Sensitive Information - (922)	NO
Identify Actors		3
6.1	Improper Certificate Validation - (295)	NO
6.2	Improper Following of a Certificate's Chain of Trust - (296)	NO
6.3	Improper Validation of Certificate with Host Mismatch - (297)	NO
6.4	Improper Validation of Certificate Expiration - (298)	YES
6.5	Improper Check for Certificate Revocation - (299)	NO
6.6	Insufficient Verification of Data Authenticity - (345)	NO
6.7	Origin Validation Error - (346)	YES
6.8	Missing Check for Certificate Revocation after Initial Check - (370)	NO
6.9	Unintended Proxy or Intermediary ('Confused Deputy') - (441)	NO
6.10	Missing Validation of OpenSSL Certificate - (599)	YES
6.11	Improper Verification of Source of a Communication Channel - (940)	NO
6.12	Incorrectly Specified Destination in a Communication Channel - (941)	NO
Limit access		3
7.1	Information Exposure Through Sent Data - (201)	NO
7.2	Information Exposure Through an Error Message - (209)	NO
7.3	Improper Cross-boundary Removal of Sensitive Data - (212)	NO
7.4	Creation of chroot Jail Without Changing Working Directory - (243)	NO
7.5	Execution with Unnecessary Privileges - (250)	NO
7.6	Externally Controlled Reference to a Resource in Another Sphere - (610)	YES
7.7	Improper Restriction of XML External Entity Reference - (611)	YES
7.8	External Control of File Name or Path - (73)	YES
Limit exposure		0
8.1	Information Exposure Through Self-generated Error Message - (210)	NO
8.2	Information Exposure Through Externally-Generated Error Message - (211)	NO
8.3	Information Exposure Through Process Environment - (214)	NO
8.4	Information Exposure Through Server Error Message - (550)	NO
8.5	Inclusion of Functionality from Untrusted Control Sphere - (829)	NO
8.6	Inclusion of Web Functionality from an Untrusted Source - (830)	NO
Lock computer		0
9.1	Overly Restrictive Account Lockout Mechanism - (645)	NO
Manage User Session		4
10.1	Session Fixation - (384)	YES
10.2	Exposure of Data Element to Wrong Session - (488)	YES
10.3	J2EE Bad Practices: Non-serializable Object Stored in Session - (579)	NO
10.4	J2EE Misconfiguration: Insufficient Session-ID Length - (6)	NO
10.5	Insufficient Session Expiration - (613)	YES
10.6	Improper Enforcement of Behavioral Workflow - (841)	YES
Validate Inputs		31
11.1	Improper Neutralization of Special Elements - (138)	YES
11.2	Improper Neutralization of Escape, Meta, or Control Sequences - (150)	YES
11.3	Improper Input Validation - (20)	YES
11.4	Acceptance of Extraneous Untrusted Data With Trusted Data - (349)	NO

11.5	Cross-Site Request Forgery (CSRF) - (352)	YES
11.6	External Control of Assumed-Immutable Web Parameter - (472)	YES
11.7	PHP External Variable Modification - (473)	NO
11.8	Deserialization of Untrusted Data - (502)	NO
11.9	Improper Link Resolution Before File Access ('Link Following') - (59)	NO
11.10	URL Redirection to Untrusted Site ('Open Redirect') - (601)	YES
11.11	Improper Restriction of Names for Files and Other Resources - (641)	NO
11.12	Improper Neutralization of Data within XPath Expressions ('XPath Injection') - (643)	YES
11.13	Improper Neutralization of Data within XQuery Expressions ('XQuery Injection') - (652)	YES
11.14	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection') - (74)	YES
11.15	Failure to Sanitize Special Elements into a Different Plane (Special Element Injection) - (75)	YES
11.16	Improper Neutralization of Equivalent Special Elements - (76)	YES
11.17	Improper Neutralization of Special Elements used in a Command ('Command Injection') - (77)	YES
11.18	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') - (78)	NO
11.19	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') - (79)	YES
11.20	Improper Filtering of Special Elements - (790)	YES
11.21	Incomplete Filtering of Special Elements - (791)	YES
11.22	Incomplete Filtering of One or More Instances of Special Elements - (792)	YES
11.23	Only Filtering One Instance of a Special Element - (793)	YES
11.24	Incomplete Filtering of Multiple Instances of Special Elements - (794)	YES
11.25	Only Filtering Special Elements at a Specified Location - (795)	YES
11.26	Only Filtering Special Elements Relative to a Marker - (796)	YES
11.27	Only Filtering Special Elements at an Absolute Position - (797)	YES
11.28	Improper Neutralization of Argument Delimiters in a Command ('Argument Injection') - (88)	YES
11.29	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') - (89)	YES
11.30	Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection') - (90)	YES
11.31	XML Injection (aka Blind XPath Injection) - (91)	YES
11.32	Improper Neutralization of CRLF Sequences ('CRLF Injection') - (93)	YES
11.33	Improper Control of Generation of Code ('Code Injection') - (94)	YES
11.34	Improper Neutralization of Special Elements in Data Query Logic - (943)	YES
11.35	Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection') - (95)	YES
11.36	Improper Neutralization of Directives in Statically Saved Code ('Static Code Injection') - (96)	YES
11.37	Improper Neutralization of Server-Side Includes (SSI) Within a Web Page - (97)	NO
11.38	Improper Control of Filename for Include/Require Statement in PHP Program ('PHP Remote File Inclusion') - (98)	NO
11.39	Improper Control of Resource Identifiers ('Resource Injection') - (99)	YES
Verify Message Integrity		2
12.1	Missing Support for Integrity Check - (353)	NO
12.2	Improper Validation of Integrity Check Value - (354)	NO
12.3	Detection of Error Condition Without Action - (390)	NO
12.4	Unchecked Error Condition - (391)	NO
12.5	Download of Code Without Integrity Check - (494)	NO
12.6	Reliance on Cookies without Validation and Integrity Checking - (565)	NO
12.7	Reliance on Obfuscation or Encryption of Security-Relevant Inputs without Integrity Checking - (649)	NO
12.8	Improper Enforcement of Message or Data Structure - (707)	YES
12.9	Improper Handling of Exceptional Conditions - (755)	YES
12.10	Improper Enforcement of Message Integrity During Transmission in a Communication Channel - (924)	NO
TOTAL	223	100

References

[1] ASM bytecode manipulation framework. https://asm.ow2.io/.

[2] Common attack pattern enumeration and classification (CAPEC). https://capec.mitre.org.

[3] CWE - Common Weakness Enumeration. https://cwe.mitre.org.

[4] CWE-640: Weak password recovery mechanism for forgotten password. https://cwe.mitre.org/data/definitions/640.html.

[5] CWE Category: sert cei c coding standards. https://cwe.mitre.org/data/definitions/1154.html.

[6] CWE Category: software fault patterns. https://cwe.mitre.org/data/definitions/888.html.

[7] CWE-FAQ. https://cwe.mitre.org/about/faq.html#A.1.

[8] CWE Top 25 Most Dangerous Software Errors. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html.

[9] CWE VIEW: Architectural Concepts. https://cwe.mitre.org/data/definitions/1008.html.

[10] CWE VIEW: Hardware Design. https://cwe.mitre.org/data/definitions/1194.html.

[11] CWE VIEW: Research Concepts. https://cwe.mitre.org/data/definitions/1000.html.

[12] CWE VIEW: Software Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[13] CWE View: Weaknesses in owasp top ten (2021). https://cwe.mitre.org/data/definitions/1344.html.

[14] Eclipse IDE, https://www.eclipse.org/ide/.

[15] Java Platform, Enterprise Edition. https://www.oracle.com/java/technologies/java-ee-glance.html.

[16] JUnit, https://junit.org/.

[17] Open Web Application Security Project. https://www.owasp.org/.

[18] OWASP Top 10 Mobile Security Risks. https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10.

[19] OWASP Top 10 Web Application Security Risks. https://owasp.org/www-project-top-ten/.

[20] Selenium Web Testing Framework, https://www.seleniumhq.org/.

[21] Web Ontology Language (OWL). https://www.w3.org/OWL/.

[22] Xtext, https://www.eclipse.org/Xtext/.

[23] Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, and Nadia Alshahwan. Automated testing for sql injection vulnerabilities: An input mutation approach. In *ISSTA'14*, pages 259–269, 2014.

[24] Chittineni Aruna and R Siva Ram Prasad. Metamorphic relations to improve the test accuracy of multi precision arithmetic software applications. In *ICACCI'14*, pages 2244–2248, 2014.

[25] Yosef Ashibani and Qusay H. Mahmoud. Cyber physical systems security: Analysis, challenges and solutions. *Computers & Security*, 68:81–97, 2017.

[26] Authors of this paper. SMRL editor executable, catalog of MRs, MT framework, experimental data. `https://sntsvv.github.io/SMRL/`.

[27] Bruce W. Ballard and Alan W. Biermann. Programming in natural language: "nlc" as a prototype. In *ACM'79*, pages 228–237, 1979.

[28] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[29] Stephanie Bayer, Thomas Enderle, Dennis-Kengo Oka, and Marko Wolf. Security crash test - practical security evaluations of automotive onboard it components. In Herbert Klenk, Hubert B. Keller, Erhard Plödereder, and Peter Dencker, editors, *Automotive - Safety & Security 2014*, pages 125–139, Bonn, 2015. Gesellschaft für Informatik e.V.

[30] Nikolai Mansourov Ben A. Calloni, Djenana Campana. Embedded Information Systems Technology Support (EISTS). Task Order 0006: Vulnerability Path Analysis and Demonstration (VPAD). Volume 2 - White Box Definitions of Software Fault Patterns. Technical report, LOCKHEED MARTIN INC FORT WORTH TX, 2011. `https://apps.dtic.mil/docs/citations/ADB381215`.

[31] Guru Bhandari, Amara Naseer, and Leon Moonen. *CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software*, page 30–39. Association for Computing Machinery, New York, NY, USA, 2021.

[32] M. Boehme, C. Cadar, and A. Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Software*, 38(03):79–86, may 2021.

[33] Matthias Büchler, Johan Oudinet, and Alexander Pretschner. Semi-automatic security testing of web applications from a secure model. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 253–262, June 2012.

[34] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[35] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), December 2008.

[36] Cristian Cadar and Martin Nowack. Klee symbolic execution engine in 2019. *International Journal on Software Tools for Technology Transfer*, 06 2020.

[37] Carnegie Mellon University. Semafortool. http://www.cs.cmu.edu/ ark/SEMAFOR/, 2017.

[38] Nelson H. Carreras Guzman, Morten Wied, Igor Kozine, and Mary Ann Lundteigen. Conceptualizing the key features of cyber-physical systems in a multi-layered representation for safety and security analysis. *Systems Engineering*, 23(2):189–210, 2020.

[39] Wing Kwong Chan, Tsong Y Chen, Shing Chi Cheung, TH Tse, and Zhenyu Zhang. Towards the testing of power-aware software applications for wireless sensor networks. In *ADA Europe'07*, pages 84–99, 2007.

[40] Wing Kwong Chan, Shing Chi Cheung, and Karl RPH Leung. Towards a metamorphic testing methodology for service-oriented software applications. In *QSIC'05*, pages 470–476, 2005.

[41] Wing Kwong Chan, Shing Chi Cheung, and Karl RPH Leung. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research*, 4(2):61–81, 2007.

[42] Danqi Chen and Christopher D. Manning. A fast and accurate dependency parser using neural networks. *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, (i):740–750, 2014.

[43] T. Y. Chen, F. Kuo, W. Ma, W. Susilo, D. Towey, J. Voas, and Z. Q. Zhou. Metamorphic testing for cybersecurity. *Computer*, 49(6):48–55, June 2016.

[44] Tsong Yueh Chen, Shing-Chi Cheung, and Siu-Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, The Hong Kong University of Science and Technology, 1998.

[45] Tsong Yueh Chen, Jianqiang Feng, and TH Tse. Metamorphic testing of programs on partial differential equations: a case study. In *COMPSAC'02*, pages 327–333, 2002.

[46] Tsong Yueh Chen, Joshua WK Ho, Huai Liu, and Xiaoyuan Xie. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC bioinformatics*, 10(1):24, 2009.

[47] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing: A review of challanges and opportunities. *ACM Computing Surveys*, 51(1), 2018.

[48] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, and Shengqiong Wang. Conformance testing of network simulators based on metamorphic testing technique. In *FORTE'09*, pages 243–248, 2009.

[49] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Taowei, and Long Lu. Savior: Towards bug-driven hybrid testing, 2019.

[50] Yuqi Chen, Bohan Xuan, Christopher M. Poskitt, Jun Sun, and Fan Zhang. Active fuzzing for testing and securing cyber-physical systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 14–26, New York, NY, USA, 2020. Association for Computing Machinery.

[51] Abdullahi Chowdhury, Gour Karmakar, Joarder Kamruzzaman, Alireza Jolfaei, and Rajkumar Das. Attacks on self-driving cars and their countermeasures: A survey. *IEEE Access*, 8:207308–207342, 2020.

[52] Marie-Catherine de Marneffe and Christopher D. Manning. Stanford Dependency Manual, 2008.

[53] Marie-Catherine de Marneffe and Christopher D. Manning. The stanford typed dependencies representation. In *Coling 2008: Proceedings of the Workshop on Cross-Framework and Cross-Domain Parser Evaluation*, CrossParser '08, page 1–8, USA, 2008. Association for Computational Linguistics.

[54] Mahdi Dibaei, Xi Zheng, Kun Jiang, Robert Abbas, Shigang Liu, Yuexin Zhang, Yang Xiang, and Shui Yu. Attacks and defences on intelligent connected vehicles: a survey. *Digital Communications and Networks*, 6(4):399–421, 2020.

[55] Junhua Ding, Tong Wu, Dianxiang Wu, Jun Q Lu, and Xin-Hua Hu. Metamorphic testing of a monte carlo modeling program. In *AST'11*, pages 1–7, 2011.

[56] Jürgen Dürrwang, Johannes Braun, Marcel Rumez, and Reiner Kriesten. Security evaluation of an airbag-ecu by reusing threat modeling artefacts. In *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 37–43, Dec 2017.

[57] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: Implementing domain-specific languages for java. *ACM SIGPLAN Notices - GPCE '12*, 48(3):112–121, 2012.

[58] Benjamin Fabian, Seda Gurses, Maritta Heisel, Thomas Santen, and Holger Schmidt. A comparison of security requirements engineering methods. *Requirements Engineering*, 15:7–40, 2010.

[59] Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. Chapter one - security testing: A survey. In *Advances in Computers*, volume 101, pages 1–51. Elsevier, 2016.

[60] Donald G. Firesmith. Security use cases. *Journal of Object Technology*, 2(3):53–64, 2003.

[61] Free Software Foundation. Debian GNU/Linux. `https://www.debian.org`.

[62] Juan Francisco Garcia, Daniel Jurjo, Fernando Mac, and Alessandra Gorla. An application of KLEE to aerospace industrial software. pages 1–7.

[63] Gareth Corfield. Metasploit for drones? Best of luck with that, muses veteran tinkerer. `https://www.theregister.com/2019/12/09/dronesploit_framework/`.

[64] M. Gerber, J. Chai, and A. Meyers. The role of implicit argumentation in nominal srl. In *NAACL'09*, pages 146–154, 2009.

[65] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199 – 220, 1993.

[66] Ralph Guderlei and Johannes Mayer. Towards automatic testing of imaging software by means of random and metamorphic testing. *International Journal of Software Engineering and Knowledge Engineering*, 17(6):757–781, 2007.

[67] Gus Gus. Raspberry Pi Webmin: A Web Interface for System Administration. `https://pimylifeup.com/raspberry-pi-webmin/`.

[68] Didier Guzzoni, Charles Baur, and Adam Cheyer. Modeling human-agent interaction with active ontologies. In *Interaction Challenges for Intelligent Assistants, Papers from the 2007 AAAI Spring Symposium*, pages 52–59, Stanford, California, USA, 2007. AAAI.

[69] Ines Hajri, Arda Goknil, Lionel C. Briand, and Thierry Stephany. Applying product line use case modeling in an industrial automotive embedded system: Lessons learned and a refined approach. In *MODELS'15*, pages 338–347, 2015.

[70] Ines Hajri, Arda Goknil, Lionel C. Briand, and Thierry Stephany. PUMConf: a tool to configure product specific use case and domain models in a product line. In *FSE'16*, pages 1008–1012, 2016.

[71] Ines Hajri, Arda Goknil, Lionel C. Briand, and Thierry Stephany. Configuring use case models in product families. *Software and Systems Modeling*, 17(3):939–971, 2018.

[72] Charles Haley, Robin Laney, Jonathan Moffett, and Bashar Nuseibeh. Security requirements engineering: A framework for representation and analysis. *IEEE Transactions on Software Engineering*, 34(1):133–153, 2008.

[73] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security'13*, pages 49–64, 2013.

[74] ISO. ISO 11898-1:2015, Road vehicles — Controller area network (CAN). `https://www.iso.org/standard/63648.html`.

[75] ISO. iso 14229-1:2020 road vehicles — unified diagnostic services (uds) - part 1: Application layer. `https://www.iso.org/standard/72439.html`.

[76] ISO. ISO 14229-1:2020, Road vehicles — Unified diagnostic services (UDS) — Part 1: Application layer. `https://www.iso.org/standard/14229.html`.

[77] Mingyue Jiang, Tsong Yueh Chen, Fei-Ching Kuo, and Zuohua Ding. Testing central processing unit scheduling algorithms using metamorphic testing. In *ICSESS'13*, pages 530–536, 2013.

[78] Dan Jurafsky and James H. Martin. *Speech and Language Processing (3rd ed.)*. Prentice Hall, 3 edition, 2017.

[79] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2nd Edition)*. Prentice-Hall, Inc., USA, 2009.

[80] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (3rd Edition)*. 2020.

[81] René Just and Franz Schweiggert. Evaluating testing strategies for imaging software by means of mutation analysis. In *ICSTW'09*, pages 205–209, 2009.

[82] T. Kasami. An efficient recognition and syntax analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.

[83] Rafiullah Khan, Kieran McLaughlin, David Laverty, and Sakir Sezer. Stride-based threat modeling for cyber-physical systems. In *2017 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*, pages 1–6, Sep. 2017.

[84] Kyounggon Kim, Jun Seok Kim, Seonghoon Jeong, Jo-Hee Park, and Huy Kang Kim. Cybersecurity for autonomous vehicles: Review of attacks and defense. *Computers & Security*, 103:102150, 2021.

[85] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, ACL '03, page 423–430, USA, 2003. Association for Computational Linguistics.

[86] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[87] F-C Kuo, Zhiquan Q Zhou, Jun Ma, and Guangquan Zhang. Metamorphic testing of decision support systems: a case study. *IET software*, 4(4):294–301, 2010.

[88] Fei-Ching Kuo, Tsong Yueh Chen, and Wing K Tam. Testing embedded software by metamorphic testing: A wireless metering system case study. In *LCN'11*, pages 291–294, 2011.

[89] Fei-Ching Kuo, Shuang Liu, and Tsong Yueh Chen. Testing a binary space partitioning algorithm with metamorphic testing. In *SAC'11*, pages 1482–1489, 2011.

[90] Mathias Landhausser, Sebastian Weigelt, and Walter F. Tichy. NLCI: a natural language command interpreter. *Automated Software Engineering*, 24(4):839–861, 2017.

[91] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices*, 49(6):216–226, 2014.

[92] Vu Le, Sumit Gulwani, and Zhendong Su. SmartSynth: Synthesizing smartphone automation scripts from natural language. In *MobiSys'13*, pages 193–206, 2013.

[93] Franck Lebeau, Bruno Legeard, Fabien Peureux, and Alexandre Vernotte. Model-based vulnerability testing for web applications. In *ICSTW'13*, pages 445–452, 2013.

[94] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10, February 1966.

[95] Linux CAN. CAN utils, utilities to manage can messages. `https://github.com/linux-can/can-utils`.

[96] Huai Liu, Fei-Ching Kuo, Dave Towey, and Tsong Yueh Chen. How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering*, 40(1):4–22, 2014.

[97] Phu X. Mai, Arda Goknil, Lwin Khin Shar, Fabrizio Pastore, Lionel C. Briand, and Shaban Shaame. Modeling security and privacy requirements: a use case-driven approach. *Information and Software Technology*, 100:165–182, 2018. Available at `https://orbilu.uni.lu/handle/10993/35498`.

[98] Phu X. Mai, Fabrizio Pastore, Arda Goknil, and Lionel Briand. Metamorphic security testing for web systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 186–197, 2020. Available at `https://orbilu.uni.lu/handle/10993/41229`.

[99] Phu X. Mai, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. A natural language programming approach for requirements-based security testing. In *ISSRE'18*, pages 58–69, 2018. Available at `https://orbilu.uni.lu/handle/10993/36301`.

[100] Phu X. Mai, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. MCP: a security testing tool driven by requirements. In *ICSE'19*, 2019. Available at `https://orbilu.uni.lu/handle/10993/39603`.

[101] Xuan Phu Mai. *Automated, Requirements-based Security Testing of Web-oriented Software Systems*. PhD thesis, University of Luxembourg, 9 2020. Available at `https://orbilu.uni.lu/handle/10993/44344`.

[102] V. M. Manes, H. Han, C. Han, S. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, nov 2021.

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

[104] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.*, 19(2):313–330, June 1993.

[105] Paul Dan Marinescu and Cristian Cadar. Katch: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 235–245, New York, NY, USA, 2013. Association for Computing Machinery.

[106] Johannes Mayer and Ralph Guderlei. On random testing of image processing applications. In *QSIC'06*, pages 85–92, 2006.

[107] Ali Mesbah, Engin Bozdag, and Arie Van Deursen. Crawling ajax by inferring user interface state changes. In *ICWE'08*, pages 122–134, 2008.

[108] Ali Mesbah, Arie Van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1):3, 2012.

[109] Matteo Meucci and Andrew Muller. OWASP Testing Guide v4. `https://www.owasp.org/images/1/19/OTGv4.pdf`.

[110] A. Meyers, R. Reeves, C. Macleod, R. Szekely, V. Zielinska, B. Young, and R. Grishman. The NomBank project: An interim report. In *HLT-NAACL'04*, pages 24–31, 2004.

[111] Microsoft Corp. Silverlight plug-ins and development tools. `https://www.microsoft.com/silverlight/`.

[112] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, dec 1990.

[113] MITRE. Mitre corporation. `https://www.mitre.org`.

[114] MITRE Corporation. Common vulnerabilities and exposures. `https://cve.mitre.org/cve/`.

[115] Haralambos Mouratidis and Paolo Giorgini. Secure tropos: a security-oriented extension of the tropos methodology. *International Journal of Software Engineering and Knowledge Engineering*, 17(2):285–309, 2007.

[116] C. Murphy, K. Shen, and G. Kaiser. Using jml runtime assertion checking to automate metamorphic testing in applications without test oracles. In *ICST'09*, pages 436–445, 2009.

[117] Christian Murphy, Gail E Kaiser, and Lifeng Hu. Properties of machine learning applications for use in metamorphic testing. Technical report, Columbia University, 2008.

[118] Christian Murphy, Mohammad S Raunak, Andrew King, Sanjian Chen, Christopher Imbriano, Gail Kaiser, Insup Lee, Oleg Sokolsky, Lori Clarke, and Leon Osterweil. On effective testing of health care simulation software. In *SEHC'11*, pages 40–47, 2011.

[119] Saahil Ognawala, Martín Ochoa, Alexander Pretschner, and Tobias Limmer. MACKE: Compositional analysis of low-level vulnerabilities with symbolic execution. In *ASE'16*, pages 780–785, 2016.

[120] Andreas L. Opdahl and Guttorn Sindre. Experimental comparison of attack trees and misuse cases for security threat identification. *Information and Software Technology*, 51:916–932, 2009.

[121] Martha Palmer, Daniel Gildea, and Paul Kingsbury. The proposition bank: An annotated corpus of semantic roles. *Computational Linguistics*, 31(1):71–106, 2005.

[122] Mauro Pezze and Cheng Zhang. Automated test oracles: A survey. *Advances in Computers*, 95:1–48, 2014.

[123] Oscar Pulido-Prieto and Ulises Juárez-Martínez. A survey of naturalistic programming technologies. *ACM Computing Surveys*, 50(5):70:1–70:35, 2017.

[124] Laura L Pullum and Ozgur Ozmen. Early results from metamorphic testing of epidemiological models. In *BioMedCom'12*, pages 62–67, 2012.

[125] Arvind Ramanathan, Chad A Steed, and Laura L Pullum. Verification of compartmental epidemiological models using metamorphic testing, model checking and visual analytics. In *BioMedCom'12*, pages 68–73, 2012.

[126] Joanna CS Santos, Anthony Peruma, Mehdi Mirakhorli, Matthias Galstery, Jairo Veloz Vidal, and Adriana Sejfia. Understanding software vulnerabilities related to architectural security tactics: An empirical investigation of chromium, php and thunderbird. In *ICSA'17*, pages 69–78, 2017.

[127] Joanna CS Santos, Katy Tarrit, and Mehdi Mirakhorli. A catalog of security architecture weaknesses. In *ICSAW'17*, pages 220–223, 2017.

[128] Ina Schieferdecker. Model-based testing. *IEEE Software*, 29(1):14–18, Jan 2012.

[129] Sergio Segura, Amador Durán, Ana B Sánchez, Daniel Le Berre, Emmanuel Lonca, and Antonio Ruiz-Cortés. Automated metamorphic testing of variability analysis tools. *Software Testing, Verification and Reliability*, 25(2):138–163, 2015.

[130] Sergio Segura, Amador Durán, Javier Troya, and Antonio Ruiz Cortés. A template-based approach to describing metamorphic relations. In *MET'17*, pages 3–9, 2017.

[131] Sergio Segura, Amador Durán, Javier Troya, and Antonio Ruiz-Cortés. Metamorphic relation template v1. 0. *Applied Software Engineering Research Group, Tech. Rep. ISA-17-TR-01*, 2017.

[132] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortes. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016.

[133] Sergio Segura, Robert M Hierons, David Benavides, and Antonio Ruiz-Cortés. Automated test data generation on the analyses of feature models: A metamorphic testing approach. In *ICST'10*, pages 35–44, 2010.

[134] Sergio Segura, Robert M Hierons, David Benavides, and Antonio Ruiz-Cortés. Automated metamorphic testing on the analyses of feature models. *Information and Software Technology*, 53(3):245–258, 2011.

[135] Sergio Segura, José A Parejo, Javier Troya, and Antonio Ruiz-Cortés. Metamorphic testing of RESTful web APIs. *IEEE Transactions on Software Engineering*, 44(11):1083–1099, 2017.

[136] Fute Shang, Buhong Wang, Tengyao Li, Jiwei Tian, and Kunrui Cao. Cpfuzz: Combining fuzzing and falsification of cyber-physical systems. *IEEE Access*, 8:166951–166962, 2020.

[137] José L. Silva, José Creissac Campos, and Ana C. R. Paiva. Model-based user interface testing with spec explorer and concurtasktrees. *Electronic Notes in Theoretical Computer Science*, 208:77–93, 2008.

[138] Matt Staats, Michael W Whalen, and Mats PE Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *ICSE'11*, pages 391–400, 2011.

[139] Chang-ai Sun, Guan Wang, Baohong Mu, Huai Liu, ZhaoShun Wang, and Tsong Yueh Chen. Metamorphic testing for web services: Framework and a case study. In *ICWS'11*, pages 283–290, 2011.

[140] Synopsys Inc. Description of the openssl heartbleed vulnerability. `http://heartbleed.com/`.

[141] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *APSEC'10*, pages 270–279, 2010.

[142] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving. *IEEE Transactions on Software Engineering*, 46(2):163–195, Feb 2020.

[143] Gu Tian-yang, Shi Yin-Sheng, and Fang You-yuan. Research on software security testing. *World Academy of Science, Engineering and Technology*, 70(69):647–651, 2010.

[144] Omer Tripp, Omri Weisman, and Lotem Guy. Finding your way in the testing jungle: A learning approach to web security testing. In *ISSTA'13*, pages 347–357, 2013.

[145] TH Tse and Stephen S Yau. Testing context-sensitive middleware-based software applications. In *COMPSAC'04*, pages 458–466, 2004.

[146] Ubuntu. Ubuntu, Linux OS. `https://ubuntu.com/`.

[147] University of Illinois. CogComp NLP Pipeline. https://github.com/CogComp/cogcomp-nlp/tree/master/pipeline, 2017.

[148] University of Saarland. Shalmaneser. http://www.coli.uni-saarland.de/projects/salsa/shal/, 2017.

[149] Mark Utting and Bruno Legeard. *Practical Model Based Testing*. Morgan Kaufmann Publishers, 2006.

[150] Margus Veanes, Colin Campbell, Wolfram Schulte, and Nikolai Tillmann. Online testing with model programs. In *ESEC/FSE'13*, pages 273–282, 2005.

[151] Chunhui Wang, Fabrizio Pastore, Arda Goknil, Lionel Briand, and Zohaib Iqbal. Automatic generation of system test cases from use case specifications. In *ISSTA'15*, pages 385–396, 2015.

[152] Xiaoyuan Xie, Joshua Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Application of metamorphic testing to supervised classifiers. In *QSIC'09*, pages 135–144, 2009.

[153] Jean-Paul A. Yaacoub, Ola Salman, Hassan N. Noura, Nesrine Kaaniche, Ali Chehab, and Mohamad Malli. Cyber-physical systems security: Limitations, issues and future trends. *Microprocessors and Microsystems*, 77:103201, 2020.

[154] Daniel H. Younger. Recognition and parsing of context-free languages in time n3. *Information and Control*, 10(2):189 – 208, 1967.

[155] Tao Yue, Lionel C. Briand, and Yvan Labiche. Facilitating the transition from use case models to analysis models: Approach and experiments. *ACM Transactions on Software Engineering and Methodology*, 22(1):1–38, 2013.

[156] Yuriy Zacchia Lun, Alessandro D'Innocenzo, Francesco Smarra, Ivano Malavolta, and Maria Domenica Di Benedetto. State of the art of cyber-physical systems security: An automatic control perspective. *Journal of Systems and Software*, 149:174–216, 2019.

[157] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book. In *The Fuzzing Book*. Saarland University, 2019. Retrieved 2019-09-09 16:42:54+02:00.

[158] Zhi Quan Zhou, ShuJia Zhang, Markus Hagenbuchner, TH Tse, Fei-Ching Kuo, and Tsong Yueh Chen. Automated functional testing of online search services. *Software: Testing, Verification and Reliability*, 22(4):221–243, 2012.

[159] H. Zhu. Jfuzz: A tool for automated java unit testing based on data mutation and metamorphic testing methods. In *TSA'15*, pages 8–15, 2015.