ITEA 3 Call 4: Smart Engineering

D3.8 Report on methods for security testing for variant and configurable systems

Project References

PROJECT ACRONYM	XIVT		
PROJECT TITLE	EXCELLENCE IN VARIANT TESTING		
PROJECT NUMBER	17039		
PROJECT START DATE	NOVEMBER 1, 2018	**PROJECT DURATION**	36 MONTHS
PROJECT MANAGER	GUNNAR WIDFORSS, BOMBARDIER TRANSPORTATION, SWEDEN		
WEBSITE	HTTPS://WWW.XIVT.ORG/		

Document References

WORK PACKAGE	WP 3: TESTING OF CONFIGURABLE PRODUCTS	
TASK	T3.4: FUZZING AND SECURITY TESTING IN CONFIGURABLE SYSTEMS	
VERSION	V 1.0	JAN 31ST, 2022
DELIVERABLE TYPE	R (REPORT)	
DISSEMINATION LEVEL	P (PUBLIC)	

Summary

This deliverable reports the XIVT developments and achievements on security testing of variant and configurable systems. The first part contains an overview of the state-of-the-art in security testing with respect to those aspects which are most relevant to the techniques investigated within the XIVT project. The second part gives an overview of the security testing methods and tools developed in XIVT. Some of these methods are associated with specific use cases, others are agnostic to them.

Table of Contents

1. Introduction

Security testing is one of the most important tasks in a software product's life cycle. More and more software-enabled products are being connected to one another and to the internet, or have other interfaces which make them vulnerable to malevolent attacks. Intruders not only include pranksters and hackers, but also organized criminals from international gangs. Therefore, it is essential to test connected software properly not only for functionality, but also for security. This is especially important when dealing with safety-critical products, which could endanger the well-being of humans. In particular, cyber-physical systems are software-based systems interacting with a physical environment, e.g., connected and autonomous vehicles. Many industrial cyber-physical systems are safety-critical. We refer to software security, when it comes to cyber-physical systems as *cybersecurity*.

This deliverable presents the methods and tools for cybersecurity which were developed in the XIVT project. The following Section 2 gives an overview of the state of the art with respect to these methods and tools. Section 3 presents the techniques that have been developed or enhanced within XIVT. Furthermore, it reports on experimental results obtained with the XIVT use cases. Section 4 concludes the deliverable.

2. State of the Art in Security Testing

This section presents the state of the art in cybersecurity. It does not intend to cover all potential issues in software security (for a comprehensive survey, consider the textbook by Gary R. McGraw or the one by Claudia Eckert): Neither is this deliverable a comprehensive survey of the recent software security literature. We rather focus on techniques which have been investigated within the XIVT project. This includes methods to detect and identify anomalies and vulnerabilities in software, different methods for test generation, and methods to repair vulnerable code.

Some methods for security analysis require access to the program's source code ("white-box") to identify vulnerabilities and correct them. Often this is not possible, in particular if the software developers and the product manufacturer belong to different organizations. Other methods do not require to access the source code ("black-box"). Such techniques can detect software faults, but they can not localize the problem in the code. Thus, the product manufacturer has to contact the software developers and wait for a patch to repair the code, or build other mechanisms to protect the product. Subsequently, we will identify methods as white-box or black-box whenever appropriate.

2.1 Anomaly Detection

Anomaly detection is a black-box method for identifying outlier behavior in software systems. In the XIVT project, we have applied anomaly detection to address the problem of cybersecurity in connected autonomous vehicles (CAVs). Cybersecurity can be defined as the strategies, practices, and techniques to mitigate security risks and ensure that a given system is protected against unauthorized and malicious actions. We use anomaly detection to identify cybersecurity threats or attacks in CAVs as it is imperative that manufacturers ensure vehicles are both safe and secure. Ensuring safety and security of self-driving or autonomous vehicles is essential because a vehicle malfunction can lead to the injury or death of the driver, passengers, or others outside of the vehicle. The potential for injury or death necessitates that the developers of CAVs ensure these systems work safely and are protected against malicious attacks before deploying them to users on public roads.

An intrusion detection system (IDS) is studied by [Don21] to detect anomalies in the Controller Area Network (CAN)-Bus traffic. They analyzed message identifier sequences for the CAN-Bus data collected from a heavy-duty truck. However, their work was a lightweight solution in terms of its hardware requirements and might not be very effective as a stand-alone tool. It is suggested to be used in conjunction with other anomaly (CAN) detection methods. In [Zho19a], they proposed an approach that fuses a deep neural network and a triplet loss network for CAN-Bus message anomaly detection. They showed the effectiveness of their approach by comparing the results with two other approaches as a reference. Although they were able to improve the performance of the proposed method, the time consumption keeps increasing with the increase of the hidden layers. A multi-labeled hierarchical classification (MLHC) learning model for the classification of attacks is proposed by [Par20]. It is based on a machine-learning algorithm to detect anomaly behaviors of the in-vehicle network. It can classify both the type and existence or absence of attacks applicable in interior communication environments of high-speed vehicles. However, their approach is developed for sequential message injection attacks which prevent fast data processing. A deep learning technique based on time series prediction for anomaly detection is proposed by [Qin21]. They used the long short-term memory (LSTM) concept with deep learning to enhance the cybersecurity of the intelligent connected vehicle. This approach can detect abnormal messages on the CAN-Bus and prevent some types of attacks such as malicious tampering. However, there was a critical issue with this method as it was not scalable to a large number of vehicles. An Anomaly Detection Framework for CAN-Bus is proposed by [Lin20] which is based on Deep Learning. The proposed approach covers three different types of message-injected attacks namely Denial of Service (DoS), fuzzy, and impersonation attacks in the CAN traffic. However, their work had a low detection rate when compared to other ML algorithms.

2.2 Penetration Testing

Modern cyber-physical systems, e.g., cars or industrial production lines, are complex and interconnected. The increased connectivity makes a vehicle more vulnerable to cyberattacks. Penetration testing is a black-box cybersecurity testing method which helps to detect vulnerabilities before hackers can.

A proper testing framework is crucial to safeguard the vehicle from potential cyber-attacks. Automotive penetration testing emulates an attack on an automotive system to expose and mitigate cybersecurity vulnerabilities that had not been considered during design. In an automotive system, penetration testing is usually conducted by a group of security testers who are very knowledgeable about the system or by an outside team who have limited knowledge about the design and analyze the security poster from a new angle. Since pen-testers need to think like a hacker, penetration testing is also known as “ethical hacking”.

Modern-day vehicles are equipped with around 150 ECUs. These ECUs are distributed all around the vehicle and communicate with each other via in-vehicle communication networks such as a CAN. However, the CAN-Bus is primarily designed to ensure reliable communication.

Therefore, the existing built-in security features of the CAN-Bus cannot prevent the CAN network from cyberattacks. Modern-day vehicles are equipped with an infotainment system that displays a lot of valuable information and access to the internet. Due to convenience, infotainment systems are allowed to communicate with other devices in the car through CAN-Buses. A malicious third party website can load malicious code into the infotainment system and, after that, gain control over the vehicle system with the help of CAN-network. In recent years, a handful of research works have been conducted on the vulnerabilities of automotive infotainment systems. Compromising an infotainment system could lead to an attack against the critical ECU's in the car, such as the brakes and engine [Keu18]. It was shown that attackers could remotely intrude into the system through the Wi-Fi interface and get the root privilege of the IVI system. Once the attacker gains access to the system with root privileges, the attacker is allowed to modify the system settings and extract user information [Smi16]. An in-depth discussion on the vulnerabilities posed by apps can be found on the Google play store [Man18], which can be downloaded and installed onto Android devices such as the automotive infotainment system.

2.3 Fault Injection

Fault injection is an acknowledged black-box technique to assess the quality of test suites. Given a (software) product and a test suite, faults are deliberately injected, and it is measured how many of these faults are detected by the test suite. Thus, this methods allows to determine the inherent *fault-detection capabilities* of a test suite, complementing other quality measures of a test suite such as code or requirements coverage, size and complexity of test cases, etc.
In software testing, faults are injected into the source or object code. Typical operations are changing a constant or an operator (e.g., 1 to 2, + to -, or ++ to - -), deleting or changing the order of statements, or exchanging one subroutine call by another one. When applying such *mutation operators*, it may be the case that the effects of several alterations counterbalance each other; e.g., changing - by + and 3 by 1 in the code fragment (y = x - 1; y = y + 3) does not yield any visible effect on the behavior of the program. Such *dead mutants* are a problem, since the test suite has no chance to detect them, while they still decrease the mutation score of the test suite. Other topics to considerate when setting up a fault injection initiative are the selection of mutation operators, the section of software elements to mutate, and the reporting of errors detected by the test suite. A survey of software fault injection techniques can be found in [NDM 16]. For model-based software development, faults are not injected into the (generated) source code, but into the models from which the software is derived. Methods for testing model-based product lines, on which the XIVT work builds, have been investigated in [LS 2017].

2.4 Fuzzing and Attack Injection

Fuzzing is an established black-box security testing technique to identify zero-day-vulnerabilities by stimulating the interface of the SUT with unexpected and invalid inputs [Ta18]. The importance of fuzzing in finding new bugs and vulnerabilities in software is clearly stated in [Fio20]. Whilst earlier approaches generated values either completely randomly or by employing different amounts of knowledge on the input types in a black-box manner, more recent grey-box approaches rely on genetic algorithms where the inputs are generated on binary level without any type information, and tests for further mutation are selected by their ability to increase the

code coverage. A survey on fuzzing techniques can be found in [Ma19]. A comprehensive lists of the identified challenges haven been published by Böhme et al. [BCR21].
One of the challenges of these black-box fuzzing approaches is discovering hard to reach code fragments that are hidden by complex conditions to detect vulnerabilities deeply nested in the code. *White-box fuzzing* approaches employ symbolic execution to effectively cover those execute paths that are hard to reach by black-box and grey-box approaches but suffers from scalability issues. Combining white-box approaches with information on datatypes, e.g., data structures or grammars, have recently been identified as a promising direction [BCR21].

There are several fuzzers in the literature for different programming language, but mainly for C/C++ programs (e.g., [AFL17], [Che18], [Kyr20], [Hol13], [Fio20], [AMN20]), where AFL [AFL17] is the most popular one. Hawkeye [Che18] combines static analysis and dynamic fuzzing for finding C/C++ vulnerabilities. VUzzer [Hal13] is a fuzzer that implements a feedback loop to help generate new inputs from old ones, with its two main components being a static analyser and a dynamic fuzzing loop. S. Karamcheti et al. [Kar18] show that sampling distributions over mutational operators can improve the performance of AFL. They also introduce Thompson Sampling, which is a bandit-based optimization to improve the mutator distribution adaptively. They focus on improving greybox fuzzing by studying the selection of the most promising parent test case to mutate. LibFuzzer is a coverage-guided, evolutionary fuzzing engine to test C/C++ software [LF18]. Another such fuzzer is honggfuzz [HF18], a security oriented, feedback-driven, evolutionary fuzzer. Alexandre et al. [Reb14] presented a way to optimize test case selection in order to increase coverage of the software under test. Grieco et al. [Gri16] developed VDiscover, a tool to predict if a test case is likely to discover software vulnerabilities by using lightweight static and dynamic features implemented via machine learning techniques. Klees et al. [Klee18] propose some guidelines to better test and evaluate fuzzing algorithms.

Injection attacks exploit a variety of vulnerabilities in CAN buses as it lacks encryption and authentication. [Jed21] proposed a mechanism for message injection attack detection, to detect and predict malicious message injections into a CAN-Bus. They used Messages-Sequence Graphs (MSGs) to detect message injection, modeling sequences of CAN messages in a time window. All of the data sets being used in this study are related to one driver and one vehicle. To evaluate the effectiveness of the proposed approach, a set of various vehicles and drivers would be needed.

A spoofing attack using a bus-off attack against an ECU is implemented in [Ieh18], which prevents ECU from getting access to the CAN-Bus. They conducted the attack in the laboratory in a simulated environment consisting of the attack hardware and ECU, and 100% ceased the transmission of the targeted ECU over the CAN-Bus. At the same time, they transmitted the spoofing messages over the CAN-Bus and the messages were not detected by the authorized transmitting and receiving ECUs. However, they did not propose any mechanism to prevent such attacks.

Several other works have been proposed based on long short-term memory (LSTM) ([Hos20] and [Thi21]) to identify the relationship between messages traversing in the CAN-Bus. However, these models are complex and cause high overheads on the ECUs. For example, a long short-term memory (LSTM)-based intrusion detection system is developed by [Hos20] for In-Vehicle CAN-Bus communications. Through injecting DoS, Fuzzing, and Spoofing into the attack-free dataset, they were able to develop an attack data set to implement the proposed approach.

2.5 Vulnerability Detection and Identification

This section summarizes the main related work in the areas of static analysis and machine learning for the detection and identification of vulnerabilities in source code of programs.

Static Analysis

Static analysis tools work by examining the source code (white-box) and looking for flaws without executing it and producing alerts about possible flaws (e.g., [Fon14], [Jon06], [Sha01], [47], [Dah14], [Med16], [Bac17]). This requires a human auditor to then evaluate its validity and confirm the vulnerability, which often requires high efforts with respect to a given project budget and schedule.
Many of these tools perform taint analysis, tracking user inputs to determine if they reach a sensitive sink (i.e., a function that could be exploited). CQUAL [Sha01] and Splint [Eva02] were the first to implement this technique (both for the C language), using two qualifiers – *tainted* and *untainted* – to manually annotate certain parts of the program (e.g., function parameters or return values) where untrusted / trusted data may flow. User inputs were followed through the code to find out if tainted data would arrive to a parameter labeled as untainted. If this happened, an alarm would be raised. Static analysis tools tend to generate many false positives and false negatives due to the complexity of coding knowledge about vulnerabilities. WAP [Med16] also does taint analysis, but aims at reducing the number of false positives by resorting to data mining, besides also correcting automatically the located bugs.
Flynn et al. [Fly18] developed and tested several classification models that predict if static analysis alerts are true positives or false positives, using a novel combination of multiple static analysis tools, features obtained from generated alerts, code base metrics and archived audit determinations. The main purpose of the authors was to create models that could automatically classify alerts as expected-true-positive, expected-false-positive, or indeterminate, based on user-specified confidence levels.
Boudjema et al. [Bou17] presented a tool based on static analysis methods, more specifically, abstract interpretation extended with security vulnerability checks to automatically detect security problems in C applications. They verify security properties by analyzing the language specification and documentation of the main language libraries. To locate vulnerabilities, they define properties related to different classes of problems, namely format string, command execution, and buffer and memory vulnerabilities. They present a detailed description of the properties and possible attacks that can be performed to exploit the vulnerabilities addressed. Although some of the proposed properties do not have many test cases, they have shown that it is possible to detect vulnerabilities in an automated way through the described properties with an acceptable number of false positives and false negatives.
Yamaguchi *et al.* [Yam14] presented a method for a more precise static analysis that explores a data structure called *code property graph*. They combine different source code representation graphs, such as abstract syntax trees (AST), control flow graphs (CFG), and program dependence graphs (PDG), in a single graph, and then query the graph to extract data flows and analyze them in order to discover vulnerabilities.

Machine Learning

Machine learning has been used to measure the quality of software by collecting a series of attributes that reveal the presence of software defects [Ari10], [Les08]. Other approaches resort to machine learning to predict if there are vulnerabilities in a program [Neu07], [Wal09], [Per15],

which is different from identifying the bugs precisely. To support the predictions they employ various features, such as past vulnerabilities and function calls [Neu07], or a combination of code-metric analysis with meta-data gathered from application repositories [Per15].
In particular, PhpMinerI and PhpMinerII predict the presence of vulnerabilities in PHP programs [Shar12], [Shar12a], [Sha13]. The tools are first trained with a set of annotated slices that end at a sensitive sink (but do not necessarily start at an entry point), and then they are ready to identify slices with errors. WAP is different because it uses machine learning and data mining to predict if a vulnerability detected by taint analysis is actually a real bug or a false alarm [Med16]. PhpMiner and WAP tools employ standard classifiers (e.g., Logistic Regression or a Multi-Layer Perceptron).
There are a few static analysis tools that implement machine learning techniques. Chucky [Yam13] discovers vulnerabilities by identifying missing checks in C language software. The tool does taint analysis to locate the checks between entry points and sensitive sinks, applies text mining to discover the neighbors of these checks, and then builds a model to see if there are checks that might be absent. Soska et al. aim to predict whether a website will become malicious in the future, before it is actually compromised [Sos14]. Scandariato et al. [Sca14] performs text mining to predict vulnerable software components in Android applications. SuSi [Ras14] employs machine learning to classify sources and sinks in the code of Android API.
Recently, deep learning has started to be applied in the vulnerability detection field [Gri16], [Rus18], [Rab18], [Zho19], [Fan19], [FMAN20], essentially in finding C/C++ bugs [Rab18], [Zho19]. VulDeePecker [20] resorts to code gadgets to represent parts of C programs and then transforms them into vectors. A neural network system then determines if the target program is vulnerable due to buffer or resource management errors.
Russell et al. [Rus18] developed a vulnerability detection tool for C/C++ based on features learning from a dataset and artificial neural network. There are very few models for finding faults in web applications [Fan19], [FMAN20], [Rab20], which follow a somewhat similar approach as those for finding C/C++ bugs.
Yamaguchi et al. [Yam11] proposed a method for assisted discovery of vulnerabilities in source code. Their goal was to create a method to make manual auditing more effective, helping and guiding the inspection of the source code. To do this, the method places the code in a vector space, so that typical API usage patterns can be determined automatically. To capture API usage patterns and to transfer these known vulnerability patterns to other pieces of code they combined static code analysis and machine learning techniques. These patterns implicitly capture the semantics of the code and allow extrapolating known vulnerabilities, identifying potentially vulnerable code with similar characteristics. This extrapolation process serves as a guide for the analyst and facilitates the inspection of the source code. Many vulnerabilities can be captured by API usage. However, there also exist cases where the code structure is more relevant for auditing.
Deep learning requires big datasets for training the models and is not able to locate and explain detected vulnerabilities. This is due to its black-box nature which hides its internal logic and makes it difficult to understand the classification operation [Shw17]. To attain that, Devign [Zho19] combines traditional graph code representations (e.g., AST and CFG) and Natural Code Sequence in a same graph. However, the approach creates an overly complicated representation of the code though. Moreover, its code and dataset are unavailable to the public and the results are questionable, considering the outdated tools they compare the model with.

2.6 Automatic Program Repair and Patching

Automatic program repair (APR) techniques lie in two families: software repair and software healing [24,29]. APR is mainly for C programs. There are a few SATs for WebApps that employ APR [Sha12], [Hua04], [Mor20], [Med16]. WAP [Med16] corrects the code by inserting fixes on it. However, it has limitations because it can produce syntactically invalid new code that cannot be executed. [Mor20] showed that there is not a single solution to correct any kind of vulnerability, even within a vulnerability class.

Klieber et al. [Kli21] presented a technique to repair a C program at the source code level against potential violations of spatial memory safety. It differs from other traditional program repair techniques by focusing on preventing a security vulnerability from being exploited by an attacker. Although many techniques already existing that can harden software against memory related vulnerabilities, many of those create dependencies on the compilers making it difficult to inspect of fine-tune the repair itself.

The analysis and transformation needed to repair a vulnerability at the source code level is most easily done at an intermediate representation. Existing approaches, however, have fundamental limitations when it comes to translating changes back to source code. This is why the technique presented by the authors tackles this challenge by first translating the intermediate representation to an abstract syntax tree level. They use a carefully designed set of transformation rules and repair transformations, where changes can then be translated back to source code level with the help of a modified clang. This approach was implemented in a tool called ACR and tested against programs with spatial memory bugs from the SPEC CPU2006 benchmarks and the Software Verification Competition. It shows good results but at the cost of a performance overhead.

Sawadogo et al. [Saw20] proposed an approach to catch security patches as part of an automatic monitoring service of code repositories. This work was motivated by the delay between the release of a security patch and its application. To differentiate between security patches and others, they used commit log and code analysis to collect data for the binary classification task. After that, they carried out a feature engineering step in which they reduced the volume of data collected only to the essential and transformed this data into numerical vectors to use in the learning algorithms. They opted for the use of a co-training algorithm because of the lack of labeled data. This proved to be the best option once the proposed approach demonstrated high precision and recall and constituted a significant improvement over the state-of-the-art.

Chen et al. [Che19] explored and developed an approach to code correction for automatically generate patches for security vulnerabilities. To achieve this objective, they used a sequence-to-sequence machine learning technique with Byte Pair Encoding that learns the mapping between two token sequences of source code. They used data collected from GitHub commits that presented the vulnerable and corrected code to create the training dataset. They chose C functions from this datase that the seq2seq algorithm could use and divided them by different sizes. Their results showed that the seq2seq algorithm performance is low, fixing general vulnerabilities. It depends significantly on the size of the inputs used in the tests. However, they proved that it is possible to fix vulnerabilities in an automated way.

Vasic et al. [Vas19] presented an approach that jointly learns to localize and repair bugs. The model classifies the program as faulty or correct, locates the bug when the program is faulty, and applies a fix to it. To solve the problem of classification, location, and repair, they used a multiheaded pointer network architecture, where one pointer head points to the faulty location and another to the location where the correction should be made. They compared a pointer

network on top of a neural network to a graph neural network and observed that their solution achieved better results. Also, they compared the joint model with an enumerative approach. The results showed that the model outperformed the enumerative approach by using a model that can predict a fix given the location of a bug. They concluded that the solution, despite its limitations, can perform well compared to some other approaches for similar purposes.
Bader et al. [Bad19] presented Getafix, an approach that aims at producing human-like corrections and, at the same time, being able to propose corrections in time proportional to what it would take to obtain static analysis results. The approach first divides a certain set of example corrections into ASTs. Then it extracts correction patterns from these ASTs, based on a new hierarchical clustering technique that produces a hierarchy of correction patterns ranging from very general to very specific corrections. Finally, given a bug to correct, Getafix finds appropriate correction patterns, classifies all candidate corrections, and suggests the main solutions for the developer. As a check during the third step, Getafix validates each suggestion through a static analyzer to ensure that the correction removes the warning. Getafix implements a simple but effective classification technique that uses the context of a code change to select the most appropriate fix for a given bug. The idea is that the tool learns from previously created corrections how to create new ones. The results showed that the tool can perform well. It accurately predicts fixes for several bugs, reducing the time developers spend to fix recurring bugs.
Schulte et al. [Sch13] proposed a technique for repairing binary programs directly, using evolutionary computation algorithms. The authors focused their efforts on embedded and mobile systems, because such systems have high resource constraints and tight coupling with its execution environment. This makes existing techniques and tools insufficient. They were able to demonstrate that assembly and binary repairing was capable of producing the same level of results as source-statement level repairs, but in a more efficient way. This shows that not only it was faster in suggesting repairs, but also needed much less disk and memory requirements.

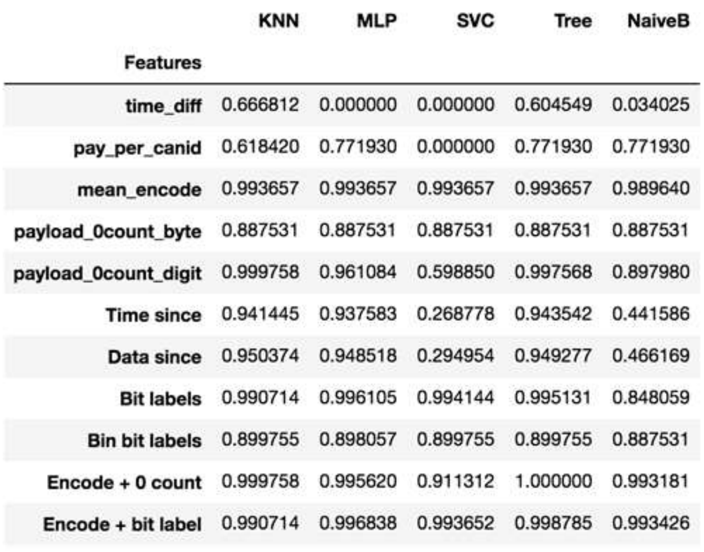
3. Methods of Security Testing used in XIVT

This section presents the tools and evaluations conducted within the XIVT project with cybersecurity methods for variants and configurable systems. The methods investigated resort to different black-box vulnerability and fault detection techniques: anomaly detection, attack and fault injection, fuzzing, and penetration testing. Furthermore, we also investigated white-box methods for vulnerability identification in source code, namely static analysis, and for automatic program repairing.

3.1 Anomaly Detection for CAV

QA Consultants have developed work on anomaly detection in CAVs which focuses on the analysis of messages sent over the CAN-Bus [Bun20]. CAN is a bus protocol with an event-triggered scheme that allows messages to be sent between a vehicle's electronic control units (ECUs). Each CAN message format is composed of a set of fields including a base ID and a data payload.

Before developing an anomaly detection method, we conducted more than 50 experiments to determine the capability of different machine learning algorithms with different feature sets to detect different types of known cyber-attacks, including denial of service (DoS) and fuzzing. For these initial experiments we utilized an existing publicly available dataset from the Hacking and Countermeasure Research Lab in South Korea [HCRL22]. For a sample of the results from these experiments see the table below, which compares the effectiveness of different features used with different machine learning algorithms to detect a fuzzing attack.



Features	KNN	MLP	SVC	Tree	NaiveB
time_diff	0.666812	0.000000	0.000000	0.604549	0.034025
pay_per_canid	0.618420	0.771930	0.000000	0.771930	0.771930
mean_encode	0.993657	0.993657	0.993657	0.993657	0.989640
payload_0count_byte	0.887531	0.887531	0.887531	0.887531	0.887531
payload_0count_digit	0.999758	0.961084	0.598850	0.997568	0.897980
Time since	0.941445	0.937583	0.268778	0.943542	0.441586
Data since	0.950374	0.948518	0.294954	0.949277	0.466169
Bit labels	0.990714	0.996105	0.994144	0.995131	0.848059
Bin bit labels	0.899755	0.898057	0.899755	0.899755	0.887531
Encode + 0 count	0.999758	0.995620	0.911312	1.000000	0.993181
Encode + bit label	0.990714	0.996838	0.993652	0.998785	0.993426

Next, we further explored the ability of the highest performing machine learning algorithms and the highest performing feature sets to detect multiple kinds of attacks as well as hybrid attacks simultaneously and then selected the best of these machine learning models as the basis for our generalized anomaly detection. To assess the efficacy of the approach on detecting previously unknown attacks we evaluated the technique on a new attack that targeted the Advanced Driver Assistant System (ADAS), described in the XIVT Pedestrian Detection System use case from Expleo.

3.2 Set of Penetration Testing for CAN-Bus and CAV

As a part of the XIVT project, and in collaboration with QA Consultants, Ontario Tech University (Ontario Tech) performed a set of penetration testing exercises on a number of automotive infotainment systems. The goal of these tests was to determine whether unauthorized access could be obtained, whether privileges could be escalated, and whether the CAN-Bus could be read or written to. The attack goal was first to attempt to access the infotainment systems remotely over wireless protocols, and in the case of failure move to physical intrusion. This plan would discover the highest risk attack vectors first, followed by testing whether CAN-Bus access was possible regardless of how the intruder got initial access to the system.

To gain insight into how security is addressed within infotainment systems, we acquired a variety of both OEM and aftermarket systems from various vendors. The following systems were used for testing:

- Pioneer AHV-2400NEX
- Pioneer MVH-2400NEX
- JVC KW-M740BT
- Auto Pumpkin Android System
- Chevrolet MyLink from 2017 Cruze
- Hyundai BlueLink from 2018 Sante Fe
- Kia UVO from 2019 Forte

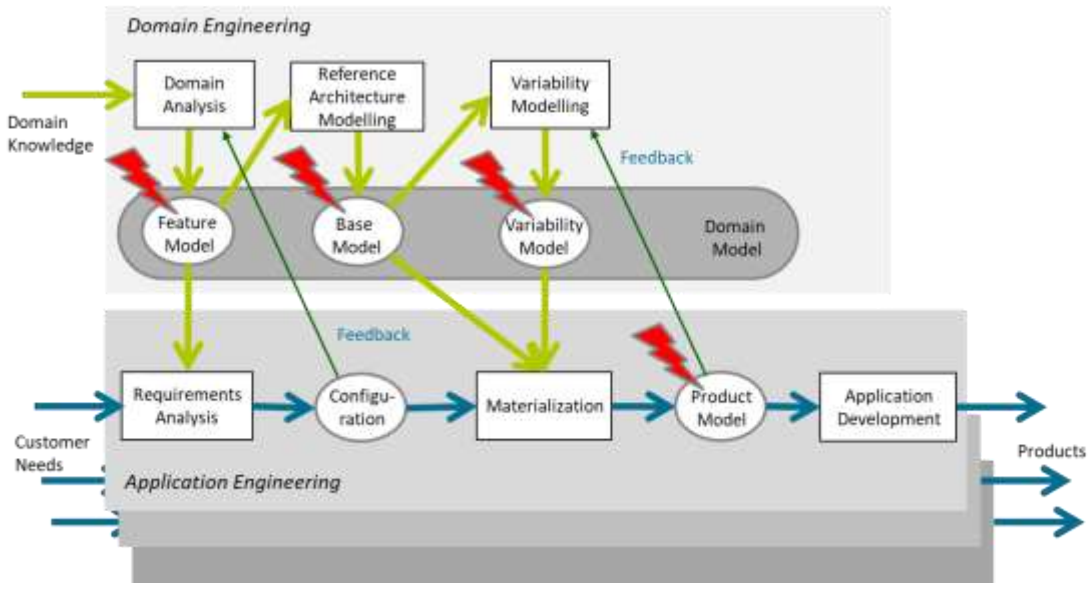
Following the pen-testing methodology, we first looked to enumerate the system, gathering information and identifying any wireless services and open ports, and then attempted to exploit any exposed services. We were unable to access any system remotely through wireless protocols, although we attempted attacks over WiFi and Bluetooth. Once it was determined that remote access was not possible, we attempted to access the systems through physical access. Although a lesser risk, physical attacks can still enable future remote access by backdooring the system. If CAN access is achieved, an attacker could potentially control the vehicle after a short period of physical access. Physical root access to the Hyundai and Kia systems was gained through debugging pins on the bottom of the unit which provide user-level interactive shell access. Using trial and error we were able to discover the correct pins and baud rate needed for the serial shell. Through further enumeration on the system, we were able to find a “Set User ID” (SUIT) binary. SUID binaries can be run by unprivileged users but with the permissions of the user with a privileged user ID. We were able to find a ***su*** binary which provided a root shell without asking for a password. This application typically is used to elevate user privileges or run privileged commands, but asks for the privileged users password; in this case it did not. However, the security of the system locked down CAN access and only provided read-only access to the file system, thereby preventing such attacks from being successful.

3.3 Attack Injection for CAN-Bus

As described in the previous subsection, QA Consultants started by focusing on the attack injection to the CAN-Bus to learn about the implication and mitigation of these attacks. Then, we worked on the three most common types of attacks, Spoof (RPM), Denial-of-service (DoS), and fuzzing, which can be performed on the CAN-Bus. In order to implement any of these attacks on the CAN-Bus, it was required to establish a communication channel. So, we evaluated several tools for communication with the CAN-Bus, including Apollo, BinFI, SAVIOR, PASTA, and CANdevStudio. We selected the CANdevStudio solution for communication with the CAN-Bus as it benefits from having a low learning curve for the user and was capable of being used with a virtual CAN-Bus.Then, we worked on the integration of the virtual CAN with the CARLA Simulator and carried out cyber-attacks. Attacks were carried out using CANdestudio to modify the vehicle's speed and steering angle and compromise the pedestrian detector's functioning through a virtual CAN-Bus. We were able to successfully implement Spoofing, Denial-of-service (DoS), and Fuzzing attacks on the CAN-Bus. This helped us to observe the implications of the attacks, which can be used to design a better mitigation mechanism in place

3.4 QATS: Quality Assessment of Test Suites

FOKUS devised and extended a tool for fault injection and assessment of test suite quality in model-based software product line development. Starting from a previous basic research prototype described in [LS2017], we developed the QATS tool for quality assessment of test suites. QATS allows to define fault injection operators on models in model-based software product line engineering. The following figure (abridged from [LS2017]) shows the MB-SPL engineering process and the possibilities which QATS offers for fault injection:



Fault injection in the feature model can be on two levels: Features can be deleted or modified, and relations between features can be altered. Whereas modifications on the first level should be detected already at the product building stage, faulty relations (e.g., changing " feature A excludes feature B" to "feature A requires feature B") might pass the product selection and product building stage, and should thus be detected during testing.

Subsequently, we describe the fault injection capabilities in more detail, following [LS2017]. For fault injection in the base model, there are several possibilities. We can omit necessary elements, add superfluous elements, or change the value of an element's attribute(s). The following mutation operators are supported by QATS for a base model formulated as a UML state machine:

- Delete Transition (DTR)
- Change Transition Target (CTT)
- Delete Effect (DEF)
- Delete Trigger (DTI)
- Insert Trigger (ITG)
- Delete Guard (DGD)
- Change Guard (CGD)

These operators are selected and implemented automatically by QATS such that they cover most of the state machine mutation operators which are found in the literature.

Mutation operators for the variability model are as follows.

Delete Mapping (DMP) The deletion of a mapping will permanently enable the mapped elements, if they are not associated to other features that constrain their enabledness otherwise. In our examples, no invalid mutants were created. However, for product lines that make heavy use of mutual exclusion (Xor and excludes) this does not apply. The reason for this are competing UML elements like transitions that would otherwise never be part of the same product. Multiple enabled and otherwise excluding transitions are possibly introducing non-determinism or at least unexpected behavior.

Delete Mapped Element (DME) This operator deletes a UML element reference from a mapping in the variability model. It resembles the case, where a modeler forgot to map a UML element that should have been mapped. Similar to the delete mapping operator, this operator may yield non-deterministic models, where otherwise excluding transitions are concurrently enabled. Product mutants equivalent to the original product model can be derived, if the feature associated to the deleted UML reference is part of the product. Again, this is results in structural equivalence to the original product.

Insert Mapped Element (IME) This operator inserts a new UML element reference to the mapping. This is the contrary case to the operators defined before, where mappings and UML elements were removed. However, inserting additional elements is more difficult than deleting them, since a heuristic must be provided for creating such an additional element. We decided to copy the first UML element reference from the subsequent mapping. If there are no more mappings, we take the first mapping. This operator is not applicable if there is just one mapping in the feature mapping model.

Swap Feature (SWP) Swapping features exchanges the mapped behavior among each other. This operator substitutes a mapping’s feature by the following mapping’s feature and vice versa. The last feature to swap is exchanged with the very first of the model. Non-deterministic behavior and thus invalid models may be designed by this operator. This is due to the fact that the mutation operator may exchange a feature from a group of mutually exclusive features by an unrestricted feature. In consequence, the previously restricted feature is now independent, while the unrestricted feature joins the mutual exclusive group. This may concurrently enable transitions which results in non-deterministic behavior. We gain structurally equivalent mutants, if the two swapped features are simultaneously activated.

Change Feature Value (CFV) This operator flips the feature value of a mapping. A modeler may have selected the wrong value for this Boolean property of each mapping. The operator must not be applied to a mapping, if there is a second mapping with the same feature, but different feature value. Otherwise, there will be two mappings for the same feature with the same feature value, which is not allowed for our feature mapping models. This operator may yield invalid mutants, if it is applied to a mapping that excludes another feature. In that case, two otherwise excluding UML elements can be present at the same time, which may result in invalid models, e.g. two default values assigned to a single variable or concurrently enabled transitions.

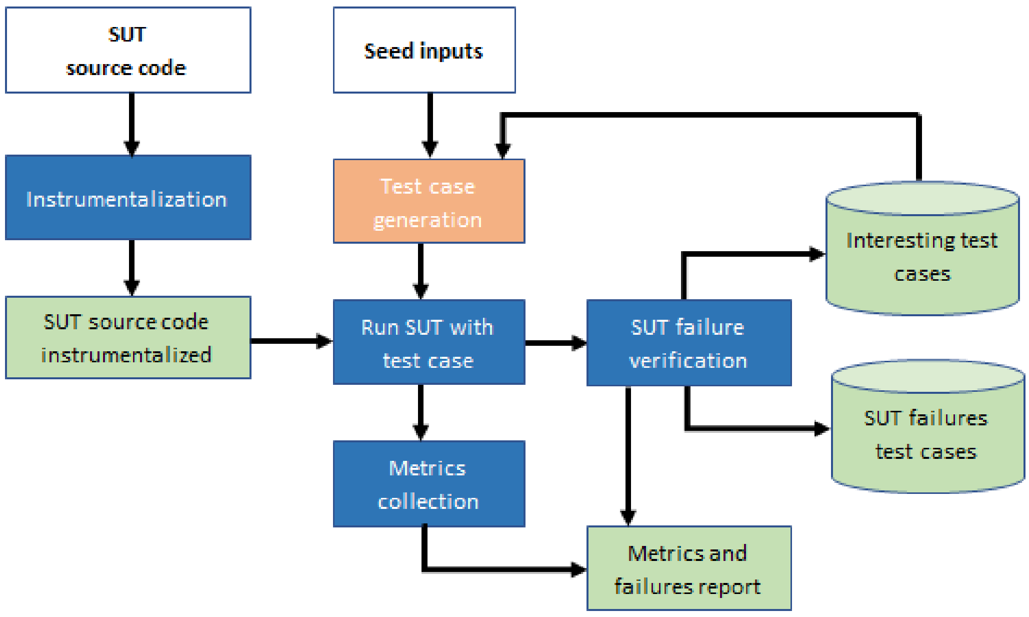
There are other possible domain model mutation operators, which, however, are of minor importance. For example, inserting superfluous mappings does not seem to be necessary: it remains unclear which and how many UML elements should be selected for the mapping. In most cases, such an operation will lead to invalid mutants.

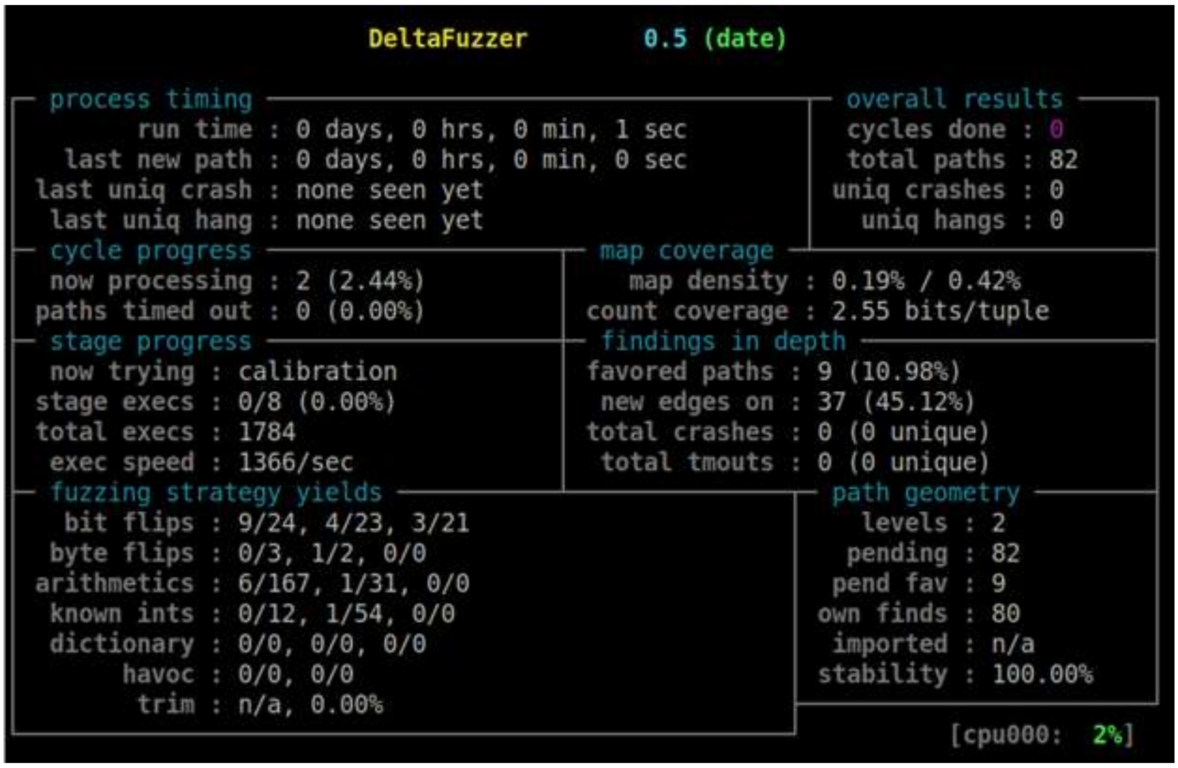
Fault injection in the product model is analogous to fault injection in the base model; the same technique and operators can be used, since the product model is derived from the base model via the materialization process.

We used QATS in a number of experiments on different academic and industrial examples. In a basic mutation experiment, each operator is used for each applicable model element exactly once. To get an idea of the complexity, consider the following numbers: For a small academic example, which has about 10 features and 44 state chart elements (states and transitions), after filtering this yields 152 product line mutants, which are resolved into 574 mutated products. Of course, it is possible to generate an arbitrarily higher amount of mutants by repeated application and combination of mutation operators. However, it is doubtful whether this brings about new insights. Thus, the test suite needs to be run on 574 products; if the test suite contains, e.g., 350 test cases and takes, e.g., 3.5 seconds for execution on a product, this results in roughly 30 minutes CPU time on an average PC for the whole campaign. Smart test case selection and test prioritization allows to bring the total number of tests to be executed in this example down to 1855, which are executed in a few seconds. For simulation-based testing, e.g., of industrial production cells where there are several collaborating robots, test execution times can be much higher; therefore, test selection and test prioritization plays a much bigger role.

3.5 DeltaFuzzer - Targeted Fuzzer

FCUL has been developing within the XIVT project the tool **DeltaFuzzer.** This is a grey box fuzzer based on the AFL fuzzer to detect several classes of vulnerabilities potentially present in software constructed in C/C++. It is the first fuzzer which implements a *Targeted Fuzzer Approach* that makes the fuzzer focus on the (novel) parts that needed to be tested and reuses knowledge acquired in previous testing campaigns. The tool is an evolution of **PandoraFuzzer** which reuses the results of testing session between variants that share same software functionalities [AMN20]. PandoraFuzzer first retrieves information about the structure of the software under test (SUT), identifying basic blocks uniquely. Next, it generates test cases for running them in the SUT, deciding which blocks are targeted. It then determines which tests are able to cause SUT failures and saves them. Afterwards, the tool calculates which saved test cases are interesting, i.e., capable of uncovering new paths and causing a SUT failure. These test cases can then be reused to generate other test cased.
The following two figures depict the architecture of DeltaFuzzer and the front-end of the tool, respectively. After the source code of the software under test (SUT) is instrumented, the tool generates a test case (randomly or through a mutation strategy of existing test cases) for running it in the SUT and collects various metrics. Next, it determines if the program suffered a failure. If this is the case, it determines the relevant test case. If the test case is "interesting", i.e., if it is capable of uncovering new execution paths and causing a SUT failure, it is saved and used later on to generate other test cases.





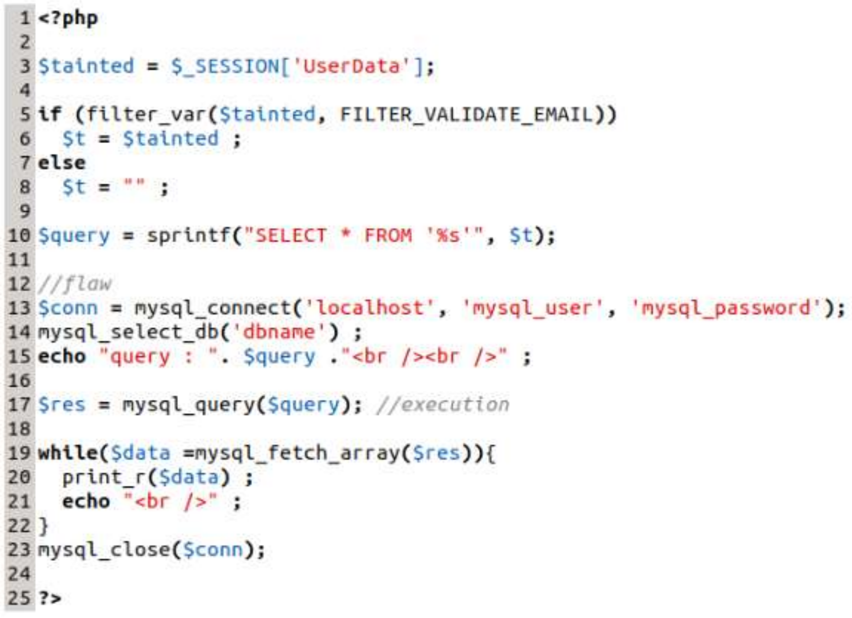
```
                      DeltaFuzzer        0.5 (date)

┌─ process timing ─────────────────────────────────────┬─ overall results ───┐
│        run time : 0 days, 0 hrs, 0 min, 1 sec        │  cycles done : 0     │
│   last new path : 0 days, 0 hrs, 0 min, 0 sec        │  total paths : 82    │
│ last uniq crash : none seen yet                      │ uniq crashes : 0     │
│  last uniq hang : none seen yet                      │   uniq hangs : 0     │
├─ cycle progress ────────────────┬─ map coverage ─┴─────────────────────┤
│  now processing : 2 (2.44%)     │    map density : 0.19% / 0.42%       │
│ paths timed out : 0 (0.00%)     │ count coverage : 2.55 bits/tuple     │
├─ stage progress ────────────────┼─ findings in depth ──────────────────┤
│  now trying : calibration       │ favored paths : 9 (10.98%)           │
│ stage execs : 0/8 (0.00%)       │  new edges on : 37 (45.12%)          │
│ total execs : 1784              │ total crashes : 0 (0 unique)         │
│  exec speed : 1366/sec          │  total tmouts : 0 (0 unique)         │
├─ fuzzing strategy yields ───────┴───────────────┬─ path geometry ──────┤
│   bit flips : 9/24, 4/23, 3/21                   │    levels : 2        │
│  byte flips : 0/3, 1/2, 0/0                      │   pending : 82       │
│ arithmetics : 6/167, 1/31, 0/0                   │  pend fav : 9        │
│  known ints : 0/12, 1/54, 0/0                    │ own finds : 80       │
│  dictionary : 0/0, 0/0, 0/0                      │  imported : n/a      │
│       havoc : 0/0, 0/0                           │ stability : 100.00%  │
│        trim : n/a, 0.00%                         ├──────────────────────┘
└──────────────────────────────────────────────────┘          [cpu000:  2%]
```

Earlier experiments with a few target libraries provided results that suggest DeltaFuzzer is able to reach most of the selected targets achieving a coverage that is higher than the traditional fuzzer solutions (e.g., AFL), with a coverage increment of 30%. Also, the initial experiments showed that it could find vulnerabilities in relatively complex software.

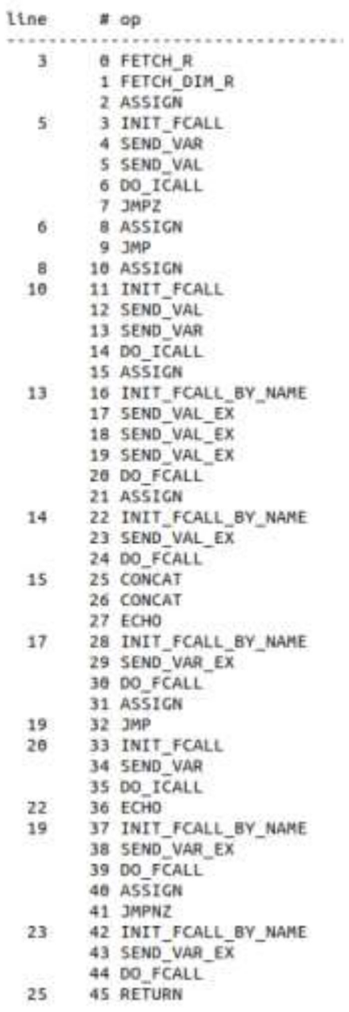
3.6 Deep Learning Model for Detecting Vulnerabilities in Web Application Variants

FCUL within XIVT also developed a **Deep Learning (DL) model** capable of classifying code excerpts of web application variants as vulnerable (or not) to SQL Injection. The model uses an intermediate language to represent the excerpts and interpret them as text, resorting to well-studied Natural Language Processing (NLP) techniques. The three figures on the next page illustrate (a) an excerpt of PHP code, (b) its representation in PHP intermediate language, and (c) the numeric vector produced from (b) which will feed the DL model.



```
 1 <?php
 2
 3 $tainted = $_SESSION['UserData'];
 4
 5 if (filter_var($tainted, FILTER_VALIDATE_EMAIL))
 6   $t = $tainted ;
 7 else
 8   $t = "" ;
 9
10 $query = sprintf("SELECT * FROM '%s'", $t);
11
12 //flaw
13 $conn = mysql_connect('localhost', 'mysql_user', 'mysql_password');
14 mysql_select_db('dbname') ;
15 echo "query : ". $query ."<br /><br />" ;
16
17 $res = mysql_query($query); //execution
18
19 while($data =mysql_fetch_array($res)){
20   print_r($data) ;
21   echo "<br />" ;
22 }
23 mysql_close($conn);
24
25 ?>
```

(a) PHP code slice



```
line     # op
---------------------------------
   3     0 FETCH_R
         1 FETCH_DIM_R
         2 ASSIGN
   5     3 INIT_FCALL
         4 SEND_VAR
         5 SEND_VAL
         6 DO_ICALL
         7 JMPZ
   6     8 ASSIGN
         9 JMP
   8    10 ASSIGN
  10    11 INIT_FCALL
        12 SEND_VAL
        13 SEND_VAR
        14 DO_ICALL
        15 ASSIGN
  13    16 INIT_FCALL_BY_NAME
        17 SEND_VAL_EX
        18 SEND_VAL_EX
        19 SEND_VAL_EX
        20 DO_FCALL
        21 ASSIGN
  14    22 INIT_FCALL_BY_NAME
        23 SEND_VAL_EX
        24 DO_FCALL
  15    25 CONCAT
        26 CONCAT
        27 ECHO
  17    28 INIT_FCALL_BY_NAME
        29 SEND_VAR_EX
        30 DO_FCALL
        31 ASSIGN
  19    32 JMP
  20    33 INIT_FCALL
        34 SEND_VAR
        35 DO_ICALL
  22    36 ECHO
  19    37 INIT_FCALL_BY_NAME
        38 SEND_VAR_EX
        39 DO_FCALL
        40 ASSIGN
        41 JMPNZ
  23    42 INIT_FCALL_BY_NAME
        43 SEND_VAR_EX
        44 DO_FCALL
  25    45 RETURN
```

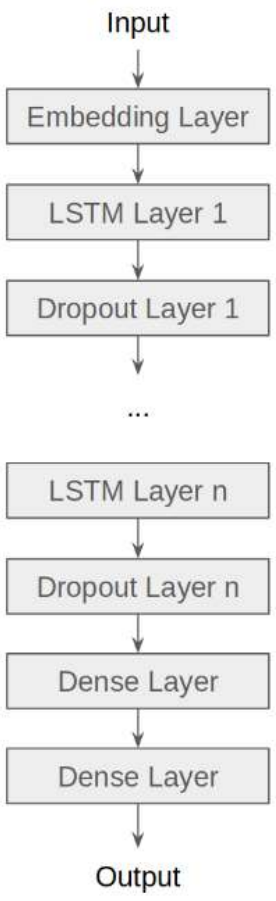
(b) Opcode slice obtained with the VLD tool

[82, 83, 40, 63, 119, 67, 131, 45, 40, 44, 40, 63, 67, 119, 131, 40, 61, 118, 118, 118, 62, 40, 61, 118, 62, 10, 10, 42, 61, 68, 62, 40, 44, 63, 119, 131, 42, 61, 68, 62, 40, 46, 61, 68, 62, 64]

(c) Numeric vector

The DL network is composed of a minimum of five layers that work sequentially. It produces a final output, between 0 and 1, indicating the probability of the excerpt being vulnerable. It receives as input a numeric vector that goes sequentially through the Embedding, LSTM, Dropout, and two Dense layers, suffering successive transformations and producing the final output. The next figure presents a high level view of the DL model we use.

We conducted experiments on four datasets of intermediate language with different excerpt representations. All datasets led to models with good performance, in which accuracy scored, on average, more than 60%. Based on these results, we can state that our DL model can help back-end programmers discover SQL Injection vulnerabilities in an early stage of the project, avoiding attacks that would eventually cost a lot to repair their damage [FMAN20].

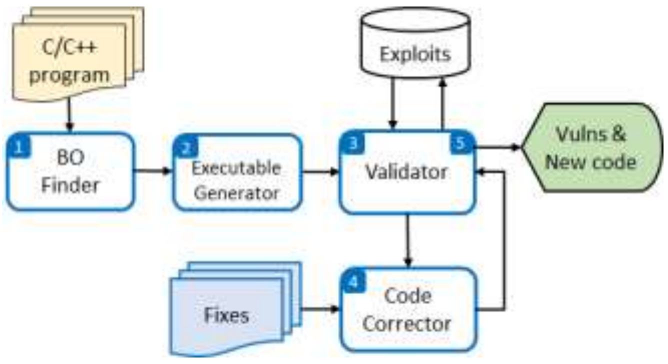


3.7 CorCA: Correction of C Automatically

FCUL within XIVT has developing **CorCA** (CORrection of C Automatically), a tool that identifies and fixes buffer overflows vulnerabilities in source code of C/C++ programs and verifies the effectiveness and correctness of the corrected (fixed) code in an automated manner.

CorCA has the goal of managing the following challenges: (1) how to find and remove vulnerabilities; (2) what is the right secure code needed to remove them; (3) where to insert this code; (4) how to keep the correct behavior of the application, after applying the code correction. The tool, to address these challenges, employs static analysis to find diverse types of buffer overflows (BO) vulnerabilities, attack injection technique to confirm the BO found and validate the fixed code, and fix the code automatically with fixes generated dynamically. The tool uses different forms to remove vulnerabilities for those cases that secure functions do not resolve.

The next figure illustrates an overview of the CorCA's architecture. Next, we present the modules that it comprises.



C/C++ Program - The C/C++ program files that we want to test and correct, which can contain one or more vulnerabilities.

BO Finder - This module is responsible for identifying possible candidate vulnerabilities in the received program. It uses static analysis techniques to collect information about potential vulnerabilities and their location in the program, namely the respective line number in the file. Using this information, it generates slices of the vulnerable code from the entry point to the sensitive sink.

Executable Generator - This module receives the vulnerable slices of code from the previous module. To generate an executable file for each candidate vulnerability found, it uses the slice received and adds from the program files other instructions needed to obtain a file which can be compiled. Then, the compiled code is instrumented, generating an executable that is forwarded to the Validator.

Validator - This module uses fuzzing techniques for validating the code received from the Executable Generator in two distinct phases. Validation is performed in the first phase to exploit the candidate vulnerabilities found by the BO Finder and generate thus the exploits for them. For those vulnerabilities it cannot exploit, they are marked as possible false positives. The remaining ones, i.e., the exploitable vulnerabilities, are signaled as such, and their exploits are stored for the second phase. The second phase uses the previously generated exploits to verify if the fixes applied are effective and safe. Also, it mutates the exploits to check if there are new exploits that can break the fixes, and that the application does not hang.

Code Corrector - This module analyzes the received code from the Validator (first phase), identifies the existing sensitive sinks, and determines the variable sizes of the arguments of the sinks. After this analysis, it checks for the possibility of buffer overflows through the size of the variables used in the sensitive sinks. If it verifies that such vulnerabilities possibly exists, it uses the fix template indicated for that sensitive sink to create a fix and applies it to the code. Also, it detects whether the code signaled possible false positive or exploitable vulnerabilities, reporting the former and proceeding with code corrections for the latter. In addition, the corrected code follows to the Executable Generator to produce its executable and then to the Validator to proceed with the second phase of validation. On this validation phase, if the code is found to be correct, as result a new release of the program is produced, with its files containing the corrected code, i.e., with the vulnerabilities fixed and corrections validated.

Vulnerabilities & New Code - The result of the procedure is a new version of the program files with the vulnerabilities fixed, and a report describing the vulnerabilities found and fixed.

CorCA was tested with C/C++ code of a software product line from Bombardier Transportation (BT), a member of the Alstom Group, and the results showed that the code does not contain BO vulnerabilities, i.e., it is safe with respect to this class of vulnerabilities.

4. Conclusions

In this report, we summarized the state of the art of various security methods for detection and identification of vulnerabilities and program repair. Also, in the second part, we described the approaches and tools/mechanisms developed within the XIVT project that implement such methods. The XIVT tools employ several security methods to cope with different scenarios. The approaches range from source code static analysis to penetration testing, including machine learning, anomaly detection, and fault injection for quality assessment of test suites. The diversity of techniques reflects the variety of characteristics in the XIVT use cases; to adequately deal with this variety required several complementary techniques. Lastly, we reported experiences and evaluations which were carried out with the XIVT tools.

5. References

[AFL17] M.Zawlewski, “AmericanFuzzyLop(AFL)Fuzzer,” http://lcamtuf.coredump.cx/afl/. 2017

[AMN20] Francisco Araújo, Ibéria Medeiros, Nuno Neves. Generating Tests for the Discovery of Security Flaws in Product Variants. In Proceedings of the International Workshop on Testing Extra-Functional Properties and Quality Characteristics of Software Systems (ITEQS), Porto, Portugal, October 2020.

[Ari10] E. Arisholm, L. C. Briand, and E. B. Johannessen, “A systematic and comprehensive investigation of methods to build and evaluate fault prediction models,” *J. Syst. Softw.*, vol. 83, no. 1, pp. 2–17, 2010

[Bac17]M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, “Efficient and flexible discovery of PHP application vulnerabilities,” in *Proc. IEEE Eur. Symp. Secur. Privacy*, Apr. 2017, pp. 334–349.

[Bad19] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages*, 2019.

[Bou17] El Habib Boudjema, Christele Faure, Mathieu Sassolas, and Lynda Mokdad. Detection of security vulnerabilities in C language applications. *Security and Privacy*, 2017.

[BCR21] Boehme, M., Cadar, C., & Roychoudhury, A. (2021). Fuzzing: Challenges and Reflections. IEEE Softw., 38(3), 79-86.

[Bun20] Arnold Buntsma and Sebastian Wilczek. Cybersecurity in automotive networks, University of Amsterdam project report. web page: https://delaat.net/rp/2019-2020/p51/report.pdf(last accessed: Jun. 4, 2020), 2020.

[Che18] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a Desired Directed Grey-box Fuzzer," inProceedings of the ACM SIGSAC Conference on Computer and Communications Security, Oct 2018, pp. 2095–2108

[Che19] Zimin Chen, Steve Kommrusch, and Martin Monperrus. Using Sequence-toSequence Learning for Repairing C Vulnerabilities. arXiv. 2019.

[Dah14]J. Dahse and T. Holz, "Simulation of built-in PHP features for precise static code analysis," in *Proc. 21st Netw. Distrib. Syst. Secur. Symp.*, Feb. 2014, pp. 23–26.

[Don21] Dönmez, Tahsin CM. "Anomaly Detection in Vehicular CAN-Bus Using Message Identifier Sequences." IEEE Access 9 (2021): 136243-136252.

[Eva02] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Softw.*, vol. 19, no. 1, pp. 42–51, Jan./Feb. 2002.

[Fan19] Y. Fang, S. Han, C. Huang, and R. Wu, "TAP: A static analysis model for PHP vulnerabilities based on token and deep learning technology," *PLoS One*, vol. 14, no. 11, Nov. 2019, Art. no. e0225196.

[Fio20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

[Fly18] Lori Flynn, William Snavely, David Svoboda, Nathan VanHoudnos, Richard Qin, Jennifer Burns, David Zubrow, Robert Stoddard, and Guillermo Marce-Santurio. 2018. Prioritizing Alerts from Multiple Static Analysis Tools, Using Classification Models. *Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies*, 13–20.

[Fon14]J. Fonseca and M. Vieira, "A practical experience on the impact of plugins in web security," in *Proc. 33rd IEEE Symp. Reliable Distrib. Syst.*, Oct. 2014, pp. 21–30.

[FMAN20] Ana Fidalgo, Ibéria Medeiros, Paulo Antunes, Nuno Neves. Towards a Deep Learning Model for Vulnerability Detection on Web Application Variants. In Proceedings of the Workshop on Testing of Configurable and Multi-variant Systems (ToCaMS), Porto, Portugal, October 2020.

[Gri16] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2016, pp. 85–96.

[Hal13] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations," inProceedings of the USENIX Security Symposium (USENIX Security 13), Aug 2013, pp. 49–64.

[HCRL22] Car-Hacking Dataset, Hacking and Countermeasure Research Lab. Web page: http://ocslab.hksecurity.net/Datasets/CAN-intrusion-dataset(last accessed: Jan 26, 2022).

[HF18]“honggfuzz,” https://github.com/google/honggfuzz/. 2018

[Hol13] Hoffman,K.L., Padberg,M., Rinaldi,G., et al.: Traveling salesman problem. Encyclopedia of operations research and management science **1**, 1573 – 1578 (2013)

[Hos20] Hossain, Md Delwar, et al. "LSTM-based intrusion detection system for in-vehicle CAN-Bus communications." IEEE Access 8 (2020): 185489-185502.

[Hua04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. 2004. Securing web application code by static analysis and runtime protection. In Proceedings of the 13th international conference on World Wide Web (WWW ’04). 40–52

[Ieh18] Iehira, Kazuki, Hiroyuki Inoue, and Kenji Ishida. "Spoofing attack using bus-off attacks against a specific ECU of the CAN-Bus." 2018 15th IEEE Annual Consumer Communications & Networking Conference (CCNC). IEEE, 2018.

[Jed21] Jedh, Mubark, et al. "Detection of Message Injection Attacks onto the CAN-Bus using Similarity of Successive Messages-Sequence Graphs." arXiv preprint arXiv:2104.03763 (2021).

[Jon06] N. Jovanovic, C. Kruegel, and E. Kirda, “Precise alias analysis for static detection of web application vulnerabilities,” in *Proc. Workshop Program. Lang. Anal. Secur.*, Jun. 2006, pp. 27–36.

[Kar18]S.Karamcheti,G.Mann,andD.Rosenberg,“AdaptiveGrey-BoxFuzz-Testing with Thompson Sampling,” inProceedings of the ACMWorkshop on Artificial Intelligence and Security, Oct 2018, pp. 37–47.

[Keu18] Keuper, D., and T. Alkemade. "‘The connected car ways to get unauthorized access and potential implications." Computest, Zoetermeer, The Netherlands, Tech. Rep (2018).

[Klee18]G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating Fuzz Testing,” inProceedings of the ACM SIGSAC Conference on Computer and Communications Security, Oct 2018, pp. 2123–2138.

[Kli21] William Klieber, Ruben Martins, Ryan Steele, Matt Churilla, Mike McCall, and David Svoboda. 2021. Automated Code Repair to Ensure Spatial Memory Safety. *Proceedings - 2021 IEEE/ACM International Workshop on Automated Program Repair, APR. pp* 23–30. 2021.

[Kyr20] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. *29th USENIX Security Symposium (USENIX Security 20)*, 2271–2287.

[Les08] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, “Benchmarking classification models for software defect prediction: A proposed framework and novel findings,” *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, Jul./Aug. 2008

[LF18] “libfuzzer,” https://llvm.org/docs/LibFuzzer.html, 2018

[Lin20] Lin, Yubin, et al. "An Evolutionary Deep Learning Anomaly Detection Framework for In-Vehicle Networks-CAN-Bus." IEEE Transactions on Industry Applications (2020).

[LS17] Lackner, H., & Schlingloff, B. H. (2017). Advances in testing software product lines. In *Advances in computers* (Vol. 107, pp. 157-217). Elsevier.

[Ma19] Manès, V. J. M., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., & Woo, M. (2019). The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering.

[Man18] Mandal, Amit Kr, et al. "Vulnerability analysis of android auto infotainment apps." Proceedings of the 15th ACM International Conference on Computing Frontiers. 2018.

[Med16] Ibéria Medeiros, Nuno Neves, Miguel Correia. Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining. IEEE Transactions on Reliability. Vol. 65, No. 1, pages 54-69, March 2016

[Mor20] Ricardo Morgado, Ibéria Medeiros, Nuno Neves. Towards Web Application Security by Automated Code Correction. In Proceedings of the International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), Prague, Czech Republic, May 2020

[NDM16] Natella, Roberto, Domenico Cotroneo, and Henrique S. Madeira. "Assessing dependability with software fault injection: A survey." ACM Computing Surveys (CSUR) 48, no. 3 (2016): 1-55.

[Neu07] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, “Predicting vulnerable software components,” in *Proc. 14th ACM Conf. Comput. Commun.Secur.*, 2007, pp. 529–540

[Par20] Park, Seunghyun, and Jin-Young Choi. "Hierarchical anomaly detection model for in-vehicle networks using machine learning algorithms." Sensors 20.14 (2020): 3934.

[Per15] H. Perl *et al.*, “VCCFinder: Finding potential vulnerabilities in opensource projects to assist code audits,” in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 426–437.

[Qin21] Qin, Hongmao, Mengru Yan, and Haojie Ji. "Application of Controller Area Network (CAN) bus anomaly detection based on time series prediction." Vehicular Communications 27 (2021): 100291.

[Rab18]R. Rabheru, H. Hanif and S. Maffeis, “VulDeePecker: A deep learning based system for vulnerability detection,” in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, Feb. 2018

[Rab20]R. Rabheru, H. Hanif, and S. Maffeis, “A hybrid graph neural network approach for detecting PHP vulnerabilities,” *ArXiv*, vol. abs/2012.08835 Dec. 2020.

[Ras14] S. Rasthofer, S. Arzt, and E. Bodden, “A machine-learning approach for classifying and categorizing android sources and sinks,” in *Proc. Netw. Distrib. Syst. Secur. Symp.*, Feb. 2014, Art. no. 1125.

[Reb14] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, “Optimizing Seed Selection for Fuzzing,” inIn Proceedings of the USENIX Security Symposium, Aug 2014, pp. 861–875

[Rus18] Rebecca L Russell, Louis Kim, Lei H Hamilton, Tomo Lazovich, Jacob A Harer, Onur Ozdemir, Paul M Ellingwood, and Marc W McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. (2018).

[Saw20] Arthur D. Sawadogo, Tegawende F. Bissyand ´ e, Naouel Moha, Kevin Allix, Jacques Klein, Li Li, and Yves Le Traon. Learning to Catch Security Patches. arXiv. 2020.

[Sca14]R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, “Predicting vulnerable software components via textmining,” *IEEE Trans. Softw. Eng.*, vol. 40, no. 10, pp. 993–1006, Oct. 2014

[Sch13] Eric Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. 2013. Automated Repair of Binary and Assembly Programs for Cooperating Embedded Devices. *SIGARCH Comput. Archit. News* 41 (3 2013), 317–328. Issue 1.

[Sha01]U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, “Detecting format string vulnerabilities with type qualifiers,” in *Proc. 10th USENIX Security Symposium.*, Aug. 2001, pp. 201–220.

[Sha12] Lwin Khin Shar and Hee Beng Kuan Tan. 2012. Automated removal of cross site scripting vulnerabilities in web applications. Inf. Softw. Technol. 54, 5 (May 2012), 467- 478.

[Sha13] L. K. Shar, H. B. K. Tan, and L. C. Briand, “Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis,” in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 642–651.

[Shar12]L. K. Shar and H. B. K. Tan, “Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities,” in *Proc.34th Int. Conf. Softw. Eng.*, 2012, pp. 1293–1296.

[Shar12a]L. K. Shar and H. B. K. Tan, “Predicting common web application vulnerabilities from input validation and sanitization code patterns,” in *Proc. 27th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2012, pp. 310–313.

[Smi16] Smith, Craig. The car hacker&apos;s handbook: a guide for the penetration tester. no starch press, 2016.

[Son11]S. Son and V. Shmatikov, “SAFERPHP: Finding semantic vulnerabilities in PHP applications,” in*Proc. ACM SIGPLAN 6th Workshop Program. Lang. Anal. Secur.*, 2011, pp. 1–13

[Sos14]K. Soska andN. Christin, “Automatically detecting vulnerablewebsites before they turn malicious,” in *Proc. 23rd USENIX Secur. Symp.*, Aug. 2014, pp. 625–640.

[Shw17]R. Shwartz-Ziv and N. Tishby, “Opening the black box of deep neural networks via information,” 2017, *arXiv:1703.00810.*

[Ta18] Takanen, A., Demott, J. D., Miller, C., & Kettunen, A. (2018). Fuzzing for software security testing and quality assurance. Artech House.

[Thi21] Thiruloga, Sooryaa Vignesh, Vipin Kumar Kukkala, and Sudeep Pasricha. "TENET: Temporal CNN with Attention for Anomaly Detection in Automotive Cyber-Physical Systems." arXiv preprint arXiv:2109.04565 (2021).

[Vas19] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural Program Repair by Jointly Learning to Localize and Repair. *International Conference on Learning Representations*, 2019.

[Wal09]J. Walden, M. Doyle, G. A. Welch, and M. Whelan, “Security of open source web applications,” in *Proc. 3rd Int. Symp. Empirical Softw. Eng.Meas.*, 2009, pp. 545–553.

[Yam11] Fabian Yamaguchi, Felix ’FX’ Lindner, and Konrad Rieck. Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities using Machine Learning. *USENIX Workshop on Offensive Technologies*, 2011

[Yam13]F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, “Chucky: Exposing missing checks in source code for vulnerability discovery,” in *Proc. 20th ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2013, pp. 499–510.

[Yam14] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” in *Proc. IEEE Symposium Security & Privacy*, May 2014, pp. 590–604.

[Zho19]Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” in *Proc. 33rd Conf. Adv. Neural Inf. Process. Syst.*, Dec. 2019, pp. 10197–10207

[Zho19a] Zhou, Aiguo, Zhenyu Li, and Yong Shen. "Anomaly detection of CAN-Bus messages using a deep neural network for autonomous vehicles." Applied Sciences 9.15 (2019): 3174.