A Supervised Machine Learning-Based Framework to Detect Low-Level Fault Injections in Software Systems

by

Aakash Anil Gangolli

A thesis submitted to the
School of Graduate and Postdoctoral Studies in partial
fulfillment of the requirements for the degree of

Master of Applied Science in Electrical and Computer Engineering

Department of Electrical, Computer, and Software Engineering

Faculty of Engineering and Applied Science

University of Ontario Institute of Technology (Ontario Tech University)

Oshawa, Ontario, Canada

August 2022

THESIS EXAMINATION INFORMATION

Submitted by: **Aakash Anil Gangolli**

Master of Applied Science in **Electrical and Computer Engineering**

Thesis title: A Supervised Machine Learning-Based Framework to Detect Low-Level Fault Injections in Software Systems

An oral defense of this thesis took place on August 08, 2022, in front of the following examining committee:

Examining Committee:

Chair of Examining Committee	Dr. Khalid Elgazzar
Research Supervisor	Dr. Qusay H. Mahmoud
Research Co-supervisor	Dr. Akramul Azim
Examining Committee Member	Dr. Sanaa Alwidian
Thesis Examiner	Dr. Kourosh Davoudi, Ontario Tech University

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

ABSTRACT

A Supervised Machine Learning-Based Framework to Detect Low-Level Fault Injections in Software Systems

Aakash Anil Gangolli

Ontario Tech University, 2022

Advisors:

Dr. Qusay H. Mahmoud
Dr. Akramul Azim

Fault injection attacks inject faults into system components, inducing abnormal software behavior. Software vulnerability analysis cannot prevent new attack vectors without software modifications. Attack detection methods utilize system-specific software features and unsupervised learning due to lack of labelled data. Unsupervised pattern recognition is vulnerable to false data injection, and Machine Learning algorithms such as Artificial and Recurrent Neural Networks are not feasible for resource-constrained software systems. Supervised detection of low-level attack effects presents a possible solution to these issues. This thesis introduces a supervised ML-based framework to detect low-level software fault injections consisting of labelled dataset generation using an instruction-level software fault injection tool to simulate attack effects. The framework is implemented on two software systems and the results demonstrate its feasibility. The thesis explores system-level threat detection due to simulated low-level attack effects and demonstrates that combining application data and software properties improves the low-level software fault injection prediction.

Keywords: attack detection; fault injection attack; low-level software fault injection; machine learning; software fault injection dataset

AUTHOR'S DECLARATION

I hereby declare that this thesis consists of original work of which I have authored. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize the University of Ontario Institute of Technology (Ontario Tech University) to lend this thesis to other institutions or individuals for the purpose of scholarly research. I further authorize University of Ontario Institute of Technology (Ontario Tech University) to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research. I understand that my thesis will be made electronically available to the public.

_____ Aakash Anil Gangolli

STATEMENT OF CONTRIBUTIONS

I hereby certify that I am the sole author of this thesis, and I have used standard referencing practices to acknowledge ideas, research techniques, or other materials that belong to others. Furthermore, I hereby certify that I am the sole source of the creative works and/or inventive knowledge described in this thesis.

Results from this thesis research have been disseminated in the following publications:

- A. Gangolli, Q. H. Mahmoud, and A. Azim, "A Machine Learning Approach to Predict System-Level Threats from Hardware-Based Fault Injection Attacks on IoT Software," *CASCON '22: 32nd Annual International Conference on Computer Science and Software Engineering*, Toronto, Canada, 2022. *(Accepted, 6 pages)*

- A. Gangolli, Q. H. Mahmoud, and A. Azim, "A Machine Learning Based Approach to Detect Fault Injection Attacks in IoT Software Systems," *2022 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Prague, Czech Republic, 2022. *(Accepted, 6 pages)*

- A. Gangolli, Q. H. Mahmoud, and A. Azim, "A Systematic Review of Fault Injection Attacks on IoT Systems," *Electronics*, vol. 11, no. 13, 2022, doi: 10.3390/electronics11132023.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my thesis supervisors Dr. Qusay H. Mahmoud and Dr. Akramul Azim, for their support and guidance throughout my graduate studies. They consistently provided me with continuous guidelines and suggestions that allowed me to develop as an independent researcher. I would also like to express my gratitude to all of the course instructors for their valuable guidance in every step of my learning stage during the masters' degree. Finally, I would like to thank my family members, and Ontario Tech University for their consistent encouragement and valuable support that has helped me to successfully complete the research.

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER 2

CHAPTER 4

CHAPTER 5

LIST OF FIGURES

CHAPTER 1

CHAPTER 2

CHAPTER 3

CHAPTER 4

CHAPTER 5

LIST OF ABBREVIATIONS AND SYMBOLS

FIA – Fault Injection Attack

SFI – Software Fault Injection

EMFI – Electromagnetic Fault Injection

NOP – Non-Operation

FLP – Flip

JMP – Jump

LLVM-IR – Low-level Virtual Machine Intermediate Representation

EFS – Embedded Fault Simulator

QEMU – Quick Emulator

SWIFI-tool – Software Implemented Fault Injection Tool

KNN – K-Nearest Neighbors

RF – Random Forest

XGBoost – Extreme Gradient Boosting

NB – Naive Bayes

GMM – Gaussian mixture model

DBSCAN – Density-Based Spatial Clustering of Applications with Noise

K-fold CV – K-fold Cross Validation

Nested-CV – Nested Cross Validation

AES – Advanced Encryption Standard

DDoS – Distributed Denial of Service

RSA – Rivest–Shamir–Adleman

MIPS – Microprocessor without Interlocked Pipelined Stages

Chapter 1

Introduction

As software systems are deployed in an increasing number of industries, such as healthcare, communication, and autonomous mobility, they become more vulnerable to various types of attacks. Software systems have a wide attack surface, and much work has been devoted towards countering Distributed Denial of Service (DDoS), Brute Force, Phishing, and other attacks. Fault injection attacks on software systems are a unique type of attack designed to inject faults into the hardware, device drivers, operating system, network components, application software, and other components of a software system [1], [2]. These attacks intend to misuse the abnormal software behavior caused by the injected faults through various sources [1]. For instance, an attacker could use an external electromagnetic field to cause voltage glitches in the Input/Output (I/O) devices of a software system, resulting in abnormal software behavior due to incorrect input data. The attackers exploit this unintentional software behavior for a variety of purposes such as unauthorized system access, data theft, and disrupting and altering the program flow causing it to crash. According to the authors of [3], such attacks have become more common and less expensive to execute on most of the personal smart devices. As fault injection attacks on software systems can have disastrous consequences, the detection and prevention of such attacks becomes critical to the overall security of the system.

The methods proposed in literature to handle fault injection attacks on software systems can be broadly divided into two categories: software vulnerability analysis and attack detection. The software vulnerability analysis techniques analyze the software to

detect any potential vulnerabilities against fault injection attacks which may cause the system to behave abnormally or crash. Software vulnerability analysis techniques evaluate the fault tolerance by purposely attacking the software system and analyzing the static and dynamic software properties utilizing techniques such as model checking and machine learning in order to detect abnormal software behavior caused by the attacks [4], [5]. Since software vulnerability analysis techniques at the compiler-level cannot be used at run-time to detect dynamically created software vulnerabilities and new attack vectors [6], attack detection methods are necessary. The attack detection methods utilize system modeling, model checking, and machine learning techniques to monitor the high-level software system properties in the live environment to detect fault injection attacks [7]–[9]. The majority of attack detection methods utilize unsupervised clustering-based machine learning techniques which may not generate the best results under unseen conditions. The vulnerability of clustering-based techniques to false data injection and adversarial machine learning can trick such detection systems into believing that abnormal data patterns are benign, which can lead to a successful attack [10]. AI methods such as Artificial Neural Networks (ANN) and Recurrent Neural Networks (RNN) [8] may produce better results and counter adversarial machine learning and false data injection, but with a larger overhead in prediction time and computing resources, which may not be feasible for some resource-constrained software systems such as cryptography, IoT, and embedded software. Other techniques utilizing dynamic symbolic execution for detecting the software effects of the attacks pose the difficulty of monitoring those software features at run-time.

Supervised machine learning techniques provide a possible solution to the above-mentioned problems as they utilize fewer computing resources than AI techniques such as RNN and perform better than unsupervised clustering-based methods. However, utilizing supervised machine learning algorithms for detection of fault injection attacks on software

systems present challenges such as generating a labelled dataset using system-specific software features. Detecting the low-level software effects of fault injection attacks can be used to address the dataset generation challenge, as well as the run-time feature monitoring challenge faced in symbolic execution techniques. An instruction-level software fault injection tool simulates the effects of fault injection attacks at the assembly instruction level of the software using various fault models to perform low-level software fault injections [6], [11]. Fault injection attacks on software systems can be countered by using an instruction-level simulation tool to generate a labelled dataset, enabling the supervised detection of low-level fault injections in a wide range of software systems. A model capable of detecting low-level software fault injections can also detect the low-level effects of fault injection attacks on software systems.

1.1 Motivation

The main motivation behind the proposed framework is to address the challenge of labelled dataset generation enabling the use of supervised machine learning to detect the low-level software effects of fault injection attacks in the live environment of a wide range of software systems. The software system-specific labelled dataset can be generated using an instruction-level software fault injection tool to simulate the attack effects on the software.

Fig. 1.1 demonstrates a clock glitch fault injection attack on a device using PIN verification software that prevents the Compare (CMP) assembly instruction from executing. This instruction skip causes an unconditional execution of the Jump (JMP) assembly instruction resulting in the validation of an invalid PIN entered by the attacker. Attackers may inject external disturbances into the clock oscillator used by the device to perform such attacks. Fig. 1.2 demonstrates an Electromagnetic Fault Injection (EMFI) attack on the input device used to enter the user PIN. Attackers may use an electromagnetic glitch injector to

cause incorrect setting of bits in the input device when the attacker enters a random incorrect PIN. For instance, if the user PIN is loaded into a register before validation, EMFI glitches may cause the register bits to flip, resulting in a modified input user PIN to the verification software. Targeted EMFI glitches injected by the attackers may result in the verification software validating the incorrect user PIN entered by the attacker.

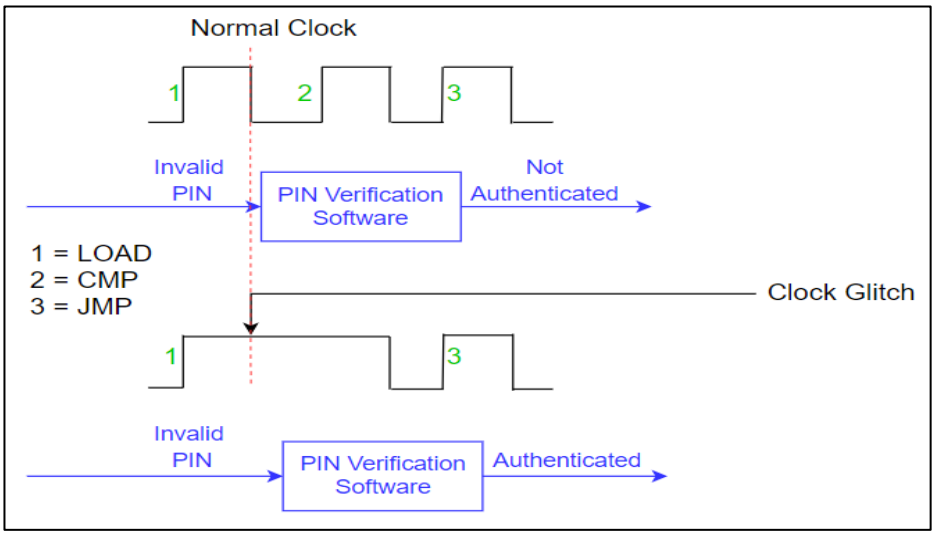

Figure 1.1: Authentication of an Invalid PIN due to Instruction Skip Caused by a Clock Glitch Fault Injection Attack.

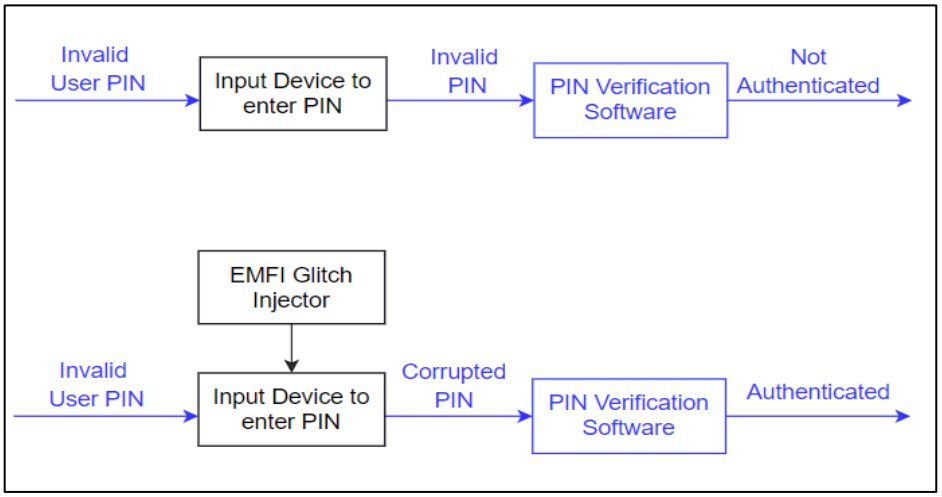

Figure 1.2: Authentication of an Invalid PIN due to Data Corruption Caused by an Electromagnetic Fault Injection Attack.

These motivating examples demonstrate that fault injection attacks affect the software system at a low-level. An instruction-level software fault injection tool [12] can be utilized to replicate such attack effects by directly manipulating the software assembly instructions [6]. Instruction-level software fault injection tools are capable of skipping instructions to simulate the effect of a clock glitch attack [6], as demonstrated in Fig. 1.1, or manipulate assembly instructions that load data into registers to simulate the effect of an EMFI attack, as demonstrated in Fig. 1.2.

This thesis investigates the use of supervised machine learning to improve the prediction of low-level attack effects on the software, and the use of a simulation tool to generate a labelled dataset enabling the inclusion of data instances for a wide range of attack scenarios. Another key motivation is detection of low-level fault injections in diverse software systems with different types of input, output and software behavior.

1.2 Contributions

The main contribution of this thesis is a supervised machine learning-based framework to detect simulated low-level fault injections in software systems, which can be used for the detection of low-level software effects of fault injection attacks in the live environment of a software system. The framework includes generation of a software system-specific labelled dataset using an instruction-level software fault injection tool, enabling the use of supervised machine learning. In this regard, two case studies on the VerifyPIN [13] and Nova Smart Home [14] software systems are conducted and the following research contributions are presented:

- A supervised machine learning-based framework for the detection of low-level software effects of fault injection attacks and its implementation on the two software systems.

- Two benchmark datasets generated by the framework for supervised detection and classification of low-level fault injections in software systems using supervised machine learning algorithms such as Random Forest and Extreme Gradient Boosting (XGBoost).
- An instruction-level software fault injection experiment conducted on the hardened version of the VerifyPIN software against fault injection attacks to demonstrate its vulnerability to low-level faults caused by fault injection attacks, necessitating low-level fault injection detection.
- Demonstrated the use of generic high-level run-time software properties such as time out, error code, and output logs to detect and classify low-level fault injections in the live environment of software systems.
- Demonstrated the accurate detection and classification of fault injection in certain types of software systems by combining application data and software properties.

1.3 Research Scope

The scope of this research primarily includes embedded and IoT software systems that utilize user input from external devices or measure the physical operating conditions of the software system using sensors to perform actions. For example, the VerifyPIN software utilizes external device to accept user PIN in smart and personal digital devices, whereas the Nova Smart Home software utilizes sensors to measure the temperature, light intensity and humidity of the home automation system. Though the proposed framework is applicable to a wide range of software systems, it is particularly useful in systems wherein different values of the application input data affects the software behavior in the presence of the same low-level fault injection. These low-level fault injections are simulated low-level software effects of fault injection attacks performed by an instruction-level software fault injection tool. Accordingly, the scope of this research includes fault injection attack vectors utilizing hardware faults that affect the assembly

instruction execution and other attack vectors that manipulate the application software input data. For example, hardware fault injections affecting the I/O devices, memory, and the application software input data are included in the scope of this research. This research employs the Non-Operation (NOP) and Flip (FLP) fault models for instruction-level software faults.

1.4 Thesis Outline

This remainder of this thesis is structured as follows.

Chapter 2 presents the relevant background information on the main concepts related to software fault injection attacks and the use of software fault models and software fault injection tools. Further on, this chapter explains the related work present in literature and state of the art methods that deal with fault injection attacks on software systems. The literature is divided into multiple categories in order to analyze them and identify their current limitations and potential solutions.

Chapter 3 provides details of the proposed framework for detecting and classifying low-level fault injections in software systems using supervised machine learning. The chapter starts by providing an overview of the proposed supervised machine learning-based framework and subsequently explains the labelled dataset generation, dataset analysis and feature selection, and machine learning stages of the framework.

Chapter 4 describes the implementation of the proposed supervised machine learning-based framework. The chapter begins by explaining in detail the Chaos Duck software fault injection tool and other tools and libraries used in the implementation. The chapter provides the implementation details of the framework, including the necessary modifications made to the chaos duck tool to implement the software system-specific dataset generation stage of the framework. Implementation of system-level threat detection due to the simulated low-level attack effects is also discussed. The chapter concludes by describing the two case

studies performed to demonstrate the implementation and evaluation of the proposed framework.

Chapter 5 presents the results obtained from the case studies. The first section of the chapter presents results obtained from the data analysis of the two labelled datasets generated in the case studies and the results obtained from the evaluation of multiple supervised machine learning models generated for the two software systems. The chapter subsequently presents the system-level threat detection results on the generated datasets. The second section of the chapter analyzes the results to compare the performance of different supervised machine learning algorithms on the two datasets and demonstrates the efficacy of the proposed framework and software system-specific labelled dataset generation.

Finally, **Chapter 6** provides the concluding remarks and offers directions for future work.

1.5 Summary

This chapter provided a preliminary discussion on fault injection attacks and the key challenges associated with the detection and classification of the attacks on software systems. In this regard, the chapter has outlined the scope and objective of this thesis, which is to enable the use of supervised machine learning algorithms for accurate and precise detection and classification of software fault injection attacks. To this end, the chapter proposes the use of an instruction-level software fault injection tool for software system-specific labelled dataset generation. Chapter 2 discusses the background on working of software fault injection tools, along with the literature and state of the art methods that deal with fault injection attacks on software-based systems and devices.

Chapter 2

Background and Related Work

This chapter provides background information on detection of software fault injection attacks by providing details on the concepts of fault injection attacks on software systems, software fault models, and software fault injection tools. In addition, this chapter also analyzes the literature and state of the art methods that deal with fault injection attacks on software systems by dividing it into multiple categories, with special focus on simulation-based methods that utilize machine learning for software vulnerability analysis and attack detection. The categorical analysis explains the need for detection of low-level fault injections in the live environment of software systems using supervised machine learning.

2.1 Definitions

This section defines the various terminologies utilized throughout this thesis.

2.1.1 Fault Injection Attacks

Fault injection attack is an attack that intends to misuse or exploit the abnormal software execution resulting from the injection of faults into various components of the software system by the attacker.

2.1.2 Low-Level Software Effects of Fault Injection Attacks

The effects of fault injection attacks observed at the software assembly instruction level are called low-level software effects of fault injection attacks. They are also termed as low-level software fault injections.

2.1.3 Instruction Level Software Fault Injection Tool

An instruction level software fault injection tool is a simulation tool used to replicate the low-level software effects of fault injection attacks by directly modifying the software assembly instructions using various software fault models.

2.1.4 Non-Operation (NOP)

NOP is a software fault model utilized by the instruction-level software fault injection tool to simulate the effects of a fault injection attack by replacing an existing software assembly instruction with an idle instruction that does nothing but consume clock cycles. NOP has been utilized in this thesis as one of the target labels in the generated datasets.

2.1.5 Flip (FLP)

FLP is a software fault model utilized by the instruction-level software fault injection tool to simulate the effects of a fault injection attack by flipping one or more bits of a software assembly instruction. FLP has been utilized in this thesis as one of the target labels in the generated datasets.

2.1.6 Jump (JMP)

JMP is a software fault model utilized by the instruction-level software fault injection tool to simulate the effects of a fault injection attack by modifying conditional and unconditional jump software assembly instructions and causing them to skip.

2.1.7 No Fault (NONE)

NONE is a software fault model utilized by the instruction-level software fault injection tool to record the software behavior under normal circumstances when there is no fault injection attack affecting the software system. NONE has been utilized in this thesis as one of the target labels in the generated datasets.

2.1.8 Nested Cross Validation (Nested-CV)

Nested-CV is an approach utilized to simultaneously train and evaluate a machine learning model by combining hyperparameter optimization with k-fold cross validation in order to select the best model and prevent overfitting, respectively.

2.1.9 System-Level Threats

A system-level threat corresponds to an unexpected or abnormal output generated by the software system due to the low-level effects of fault injection attacks. The abnormal output indicates abnormal software behavior due to the attack.

2.2 Fault Injection Attacks on Software Systems

Fault injection attacks on software systems affect the software functioning and cause it to behave abnormally [1], [6], [15]. For instance, such faults may cause the software to skip instructions, alter decision making and conditional program instructions or in the worst-case scenario, a system crash [16]. Attackers may also use these software flaws for unintended purposes such as data theft, bypassing critical authentication and security checks and disrupting the entire system [15]. There are various possible attack sources and vectors to inject faults into a software system. Fig. 2.1 depicts the various ways in which fault injection attacks can be directed at software systems. The hardware components of a system can be used to inject faults into the software. These attack vectors violate the system's assumption that hardware faults have no effect on the software behaviour [6]. However, faults introduced into various hardware components such as the external clock generator, I/O devices connected to the system, network components, and others, can alter the software behavior or cause it to crash [1], [6], [17].

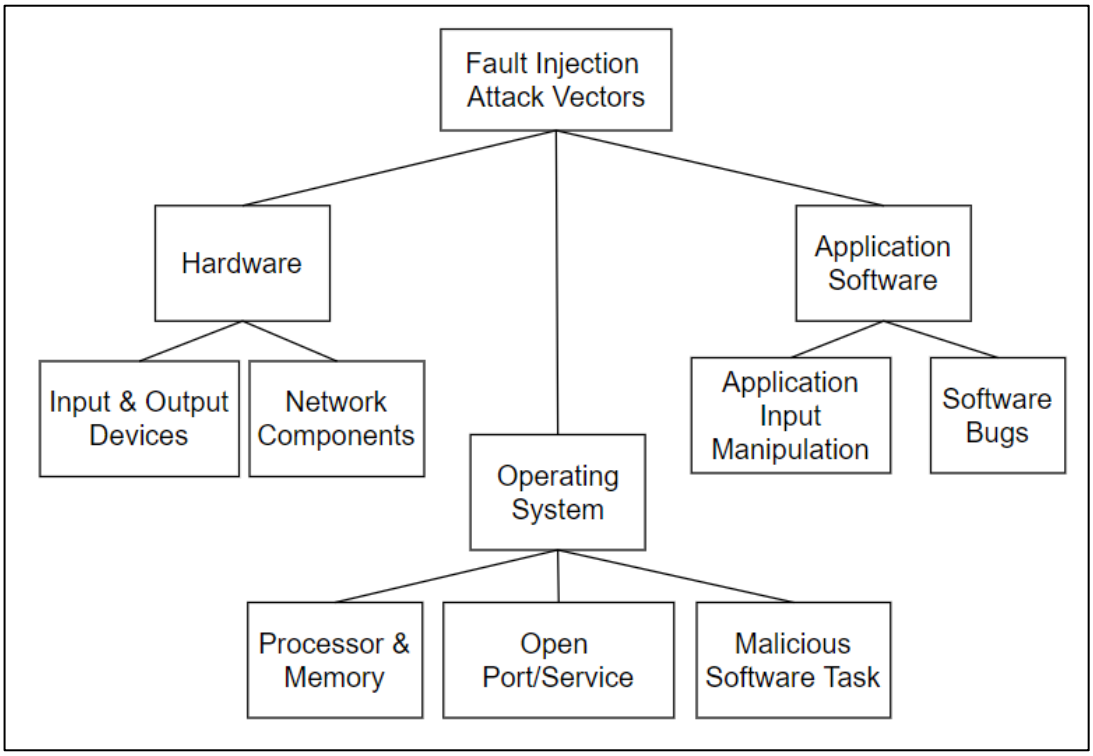

Figure 2.1: Fault Injection Attack Vectors on Software Systems.

For example, glitches can be introduced into the external clock generator of a system which modifies the falling and rising edges of the control signals used by Integrated Circuits (ICs), causing propagation delays in logic blocks [18]. This introduces faults into the instruction execution of the processor, altering the software execution behavior. Another possible attack vector is the Operating System (OS). The OS manages the interaction between system components such as the memory, I/O ports, software tasks, amongst many other. Attackers can inject faults into the OS in a variety of ways, including forcing the OS to schedule malicious software tasks and misusing open ports and services of the system. The malicious software tasks may disrupt the system by causing buffer overflows or changes to memory sections. Malicious software tasks may also starve other tasks and processes in the system of processor resources.

The application software of the system may also be a source of fault injection. These attack vectors target the application software dependencies such as the input data. The input data used by the application software may be manipulated in order to inject faults directly into the software. The application software may contain bugs that may be exploited by the injected faults. For example, attackers might misuse a known Structured Query Language (SQL) injection bug present in the software to alter the software behavior or steal confidential data. A high-level threat model is specific, whereas a low-level threat model, such as the processor instruction-level, is generic and applicable to a wide range of attack vectors.

2.3 Software Fault Models

Software fault models are created by modelling the effects of fault injection attacks on the software [11]. Software fault models can be created at multiple levels. The high-level software fault models are created by classifying the effects of fault injection attacks on the source code [19], [20], [21]. Carlo et al. [19] described various software fault models at the source code level. For instance, the control flow error model introduces faults in the conditional statements of the source code. This enables testing the software behavior for various control flow faults and deciding on the software countermeasures accordingly. The variable modification fault model changes the values of the variables used in the source code. This enables testing the software behavior for various data corruption faults which can be used to harden the source code against fault injection attacks. Using user-defined Boolean types for variables instead of 0 and 1 is an example of a countermeasure that can be used to safeguard the software against data corruption faults.

The low-level software fault models are created by classifying the effects of fault injection attacks at the program instruction level [6], [11]. Instruction-level software fault

models specify the exact assembly program instruction to be modified and the type of modification to be induced in that instruction. Instruction level software fault models enable the simulation of the effects of fault injection attacks on software systems for a variety of attack sources and vectors. Different levels at which faults can be induced in the program instructions include the bit-level, byte-level and the word-level [12]. The bit-level fault models simulate the attack by modifying one or more bits of the program instructions. For example, the single bit flip model modifies the program instructions one bit at a time by modifying 0 to 1 and 1 to 0 [11]. Fig. 2.2 shows an example of a single bit-flip fault model wherein the fourth bit changes from 1 to 0. Similarly, byte-level fault models simulate the attack by modifying a complete byte of program instruction [11], [22].

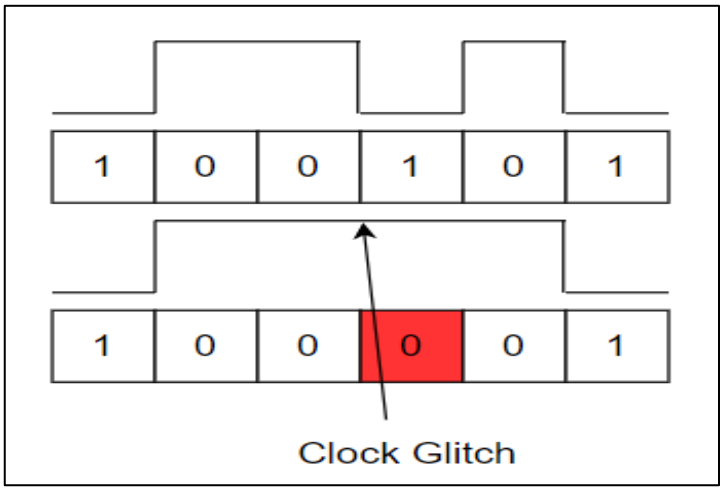

Figure 2.2: Single Bit-Flip Software Fault Model.

For example, the Non-Operation fault model (NOP) simulates the attack on the software by replacing an entire program instruction with a non-operation instruction [11], [20], [23]. The zero-byte model sets an entire byte of program instruction to zero [11]. The zero-byte and NOP fault models are used to simulate data corruption on the software. The jump fault model is another example of a byte-level model which modifies an entire byte in conditional and unconditional jump program instructions [6], [20]. The jump fault model is

used to simulate control flow modifications on the system software. Fig. 2.3 shows an example of a jump fault in the software causing the code to skip the authentication block. This causes the software to authenticate a user with an invalid PIN. A word-level fault model modifies an entire word of program instructions at once [6], [11]. This fault model is excluded from the scope of this thesis because it is primarily used to simulate error and exception conditions caused by corrupt program instructions.

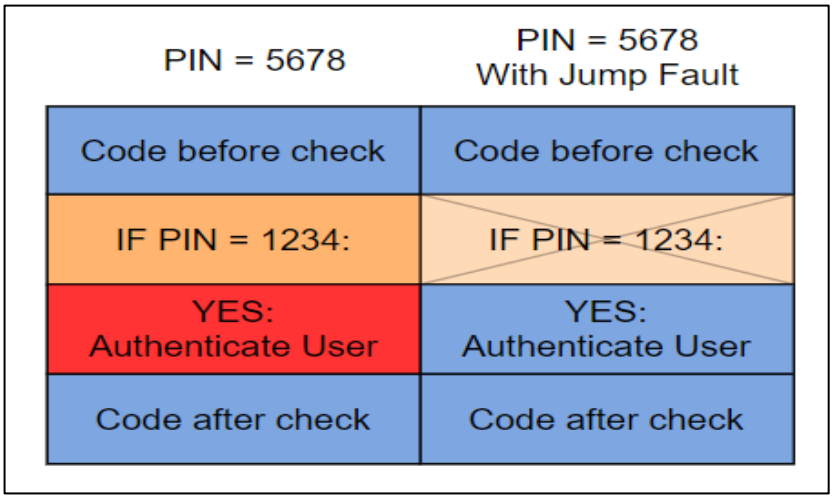

Figure 2.3: Bypassing Authentication Due to Jump Fault.

2.4 Software Fault Injection Tools

Software fault injection tools are used to implement software fault injection on different types of software systems and evaluate the resistance to a successful fault injection attack. The effects of fault injection attacks are simulated on the software using various software fault models [6]. Software fault injection tools are utilized as generic tools to test different types of software systems without having to develop a fault injection framework for individual applications. For instance, Jha et al. [24] presented the DriveFI framework used to test the effect of fault injection attacks on an autonomous vehicle system by modifying the software variable states and corrupting certain software modules. However, the utilized

framework is limited to the specific autonomous driving software architecture.

On the other hand, generic software fault injection tools with system specific modifications can also be utilized to test such software systems. Static software fault injection tools inject software faults at compile-time prior to software execution, whereas dynamic tools inject software faults at run-time during software execution. Software fault injection tools can also be classified depending on the level of fault injection. High level tools inject software faults into the source code, whereas low level tools inject faults at the program instruction level. Domenico et al. [25] presented a static high-level fault injection tool for python software called ProFIPy. Such high-level tools are restricted to specific programming languages as they are source-code dependent.

Chaos duck [12] is a dynamic low-level software fault injection tool that is capable of injecting faults into the software executable for various instruction set architectures. It performs fault injection at an instruction level on the software executable during runtime. The instruction sets currently supported by the tool include x86-32, x86-64, and arm, which are sufficient for most of the software systems in scope. A new instruction set can be added to the tool if required. Low-level tools can be used to perform fault injection on a variety of software systems irrespective of the programming language used in the source code.

2.5 Related Work

This section provides details on the relevant literature and state of the art methods that deal with fault injection attacks on software systems, with a particular emphasis on simulation-based methods that utilize machine learning for software vulnerability analysis and fault injection detection. The literature is divided into three distinct categories in order to systematically analyze them and identify the current limitations. Table 2.1 summarizes and

compares the state-of-the-art methods from each category proposed in the literature to counter fault injection attacks on software systems.

Table 2.1. Comparison of State-of-the-art Methods to Counter Fault Injection Attacks on Software Systems.

Existing Work	Type	Details
Jiang et al. [9]	Software-Based Attack Detection	1. Fault attack detection for AES software. 2. Detection considered as a system-level optimization problem using evolutionary algorithms. 3. Utilizes error detection codes.
Riviere et al. [26]	Simulation-Based Vulnerability Analysis	1. Combines Lazart and EFS for complete analysis. 2. Lazart for high level attack simulation and symbolic execution to analyze control flow graphs. 3. EFS for low-level fault simulation.
Koylu et al. [27]	Simulation-Based Fault Injection Detection	1. RNN-based program flow alteration detection due to injected faults in RSA cryptography software. 2. Simulation using instruction-level faults.
Khosrowjerdi et al. [28]	Simulation-Based Fault Injection Detection	1. Finite State Machine used to build software models. 2. QEMU-based software fault injection for simulating the attacks. 3. Utilizes automaton machine learning on reverse engineered system models for fault injection detection.
Proposed Framework	Simulation-Based Fault Injection Detection	1. Simulation using an instruction-level software fault injection tool. 2. Supervised machine learning-based framework to detect low-level software fault injections, including the creation a labelled dataset for any software system.

2.5.1 Software-Based Attack Detection

Software-based attack detection is the first category of solutions proposed in the literature. These methods utilize system modelling and model checking techniques to monitor the software system or software modifications to detect fault injection attacks. Pasqualetti et al. [7] proposed monitoring software systems by developing mathematical modelling frameworks. They designed monitoring frameworks for embedded software systems using tools from geometric control theory and mathematical modelling. These frameworks can be used for detecting fault injection attacks on centralized as well as distributed software systems. The authors of [9], [29] proposed changes in the software to detect fault injection attacks. Wei et al. [9] proposed formulating the detection of fault injection attacks as a system design problem. They recommended the use of fault detection schemes during the initial system design and implemented several fault detection schemes for confidentiality-critical software systems such as encryption and decryption algorithms. Similarly, Karpovsky et al. [29] introduced error detecting codes in the implementation of the Advanced Encryption Standard (AES) encryption algorithm used in smart cards and detected the software level effects of fault injection attacks. These methods are static and require changes to the software source code. The possibility of the attackers targeting the detection schemes implemented in the software may leave the system vulnerable to fault injection attacks.

2.5.2 Simulation-Based Vulnerability Analysis

The second category of solutions for dealing with fault injection attacks utilizes simulation of fault injection attacks on the software system in order to detect software vulnerabilities. Software fault injection tools are used to simulate the attack effects by injecting faults directly into the software [6]. Höller et al. [30] presented a Quick Emulator (QEMU) based

software fault injection platform supporting majority of the commercial processor architectures used in embedded systems [6]. This QEMU based simulator was used to simulate fault injection attacks by injecting faults at a microarchitectural-level into the emulated microprocessor components such as registers and the execution unit. The simulated faults were utilized to analyze software vulnerabilities in a password checking procedure running on an emulated microprocessor against multiple simulated fault injection. The simulated attacks were used to implement control flow modifications and memory faults.

Lacombe et al. [31] utilized static source code analysis and detected fault injection points in the software and subsequently used dynamic symbolic execution to validate the attack paths [6]. The prioritization of fault injection points using static analysis saved a significant amount of the time required for the time-consuming dynamic symbolic execution, which may also be non-terminating [6]. These non-terminating conditions were identified and avoided using the static analysis step [6]. Khoshavi et al. [32] presented Fiji-FIN, a framework to assess the fault tolerance of software systems by inducing bit perturbation attacks and soft errors. Zhang et al. [33] presented a dynamic fault injection tool which utilized the GNU debugger to inject faults without recompiling the software. Padmanabhuni et al. [5] performed static code analysis on x86 software executables and captured static properties of vulnerable program statements and learnt those patterns using unsupervised clustering-based machine learning.

Potet et al. [34] presented the Lazart tool which simulated fault injection attacks on software systems using formal methods in order to analyze the software fault tolerance. Lazart simulated fault injection attacks by operating at the Low-level Virtual Machine Intermediate Representation (LLVM-IR) [35] of the software [6]. The mutant executables generated by modifying the LLVM-IR instructions were executed using symbolic execution [6]. The symbolic executions determined the attack paths or were sometimes inconclusive.

Riviere et al. [36] combined the intermediate-level Lazart tool with low-level attack simulation utilizing the Embedded Fault Simulator (EFS) [37] to inject hardware-level faults. Lazart operated statically at compile time on the intermediate instruction representations, whereas EFS operated dynamically at run time on the assembly level instructions [6].

These simulation-based software vulnerability analysis methods are hardware-specific as the majority of them are designed for specific processor architectures or use intermediate program representation such as LLVM-IR [35]. Papadimitriou et al. [38] demonstrated that even though the AES cryptographic implementations can be secured by plugging the software vulnerabilities to some extent, they remain vulnerable to fault injection attacks. This makes it very important to detect fault injection attacks in the live environment of software systems and not rely solely on the software countermeasures.

2.5.3 Simulation-Based Fault Injection Detection

The third category of solutions extends the second category and utilizes simulation of fault injection attacks on the software systems to generate machine learning models. The machine learning models are subsequently used for the detection of fault injections in the live environment of software systems. Khosrowjerdi et al. [28] proposed a learning-based fault injection detection method. The proposed method combined machine learning and model checking of the software system to detect the effects of simulated fault injection attacks. The method used reverse engineering on system models to generate automated test cases, which were subsequently used to train an unsupervised automaton machine learning model for fault injection detection.

Koylu et al. [27] utilized an RNN-based framework to detect the program flow alterations caused in the RSA cryptography software due to the injected instruction-level

software faults [6]. An RNN was trained on the control flow graphs generated by symbolic execution of fault injected executables to detect program flow alterations due to the injected faults [6]. Yamaguchi et al. [39] proposed a framework for training unsupervised clustering machine learning models on automatically inferred search patterns in embedded software for taint-style vulnerabilities.

Given-Wilson et al. [11] described a framework for run-time verification of fault-injected software binaries using model checking to validate the expected software properties defined as variables and constants at certain positions in the software [6]. The fault-injected software binaries were created using instruction-level fault models, such as NOP and FLP [6]. Model checking applied on the fault injected binary execution classified the injection instances as vulnerable, correct, incorrect, or crashed [6].

SimpliFI [40] and SymPLFIED [41] utilized low-level fault injection simulation at the microarchitectural-level. These simulation tools require knowledge about the microprocessor architecture to simulate the attacks. For instance, SymPLFIED is a program level fault injection detection framework that uses model checking techniques and symbolic execution to detect fault injections. This architecture-dependent simulation tool is currently restricted to the Microprocessor without Interlocked Pipelined Stages (MIPS) architecture. Jha et al. [24] presented DriveFI, a framework to inject faults into the software components of an autonomous vehicle. The software behavior due to the injected faults was used to train machine learning models, which were then used for detection of fault injections in the software representing simulated fault injection attacks. The majority of the methods in this category have utilized unsupervised or semi-supervised machine learning techniques due to the lack of a labelled dataset.

2.5.4 Limitations

Due to the above-mentioned limitations of software-based attack detection methods, software fault tolerance analysis against fault injection attacks becomes necessary to implement software level countermeasures. However, these countermeasures can be breached, leaving the system vulnerable to fault injection attacks [38]. This necessitates an approach to counter fault injection attacks on software systems by using simulation-based methods to detect the attack effects on the software. The software fault injection tools simulate the attack effects by injecting faults directly into the software. However, the majority of the existing methods that utilize simulation-based approaches to detect fault injections in software systems have utilized unsupervised and semi-supervised machine learning techniques because it is cumbersome and sometimes impossible to create a labelled dataset [27], [39].

Unsupervised machine learning techniques may not produce the best results for unseen data instances encountered in the live environment of a software system. Moreover, they are vulnerable to false data injection and adversarial machine learning due to the lack of a labelled dataset to distinguish between valid and invalid data instances [6], [42]. Adversarial machine learning can trick such machine learning based detection systems into believing that abnormal data patterns are benign, which can lead to a successful attack [43]. Advanced AI techniques such as RNN used in the literature [27] may not be feasible for resource-constrained software systems and thus cannot provide a generic solution to all software systems.

These limitations present the need for utilizing supervised machine learning techniques which can be combined with techniques such as adversarial training with perturbation or noise [44], gradient masking [45] and input regularization [46] to counter adversarial machine learning [6]. However, utilizing supervised machine learning presents

the challenge of generating a labelled dataset using system-specific software features. The low-level simulation of fault injection attacks can be used to address this challenge because it can be performed on a wide range of software systems and does not depend on the source code or other system-specific features. Much of the literature has utilized instruction-level simulation tools on safety-critical software, such as cryptography software, password-checking procedures, and device firmware [6], [17], [47]–[50]. However, fault injection attacks can also affect the application software of an IoT system [6]. There is a lack of studies that utilize software fault injection tools to identify threats to application-level software, such as the control daemon of a home automation software [6]. Instruction-level software fault injection tools can be used to detect system-level threats due to low-level fault injections [6].

This thesis presents a framework to detect low-level fault injections in software systems using supervised machine learning and addresses the challenge of creating a software system-specific labelled dataset using an instruction-level software fault injection tool. The proposed framework is a hybrid method because it enables supervised low-level fault injection detection using an instruction-level software fault injection tool that is typically employed for software vulnerability analysis. Furthermore, since the behavior of certain software systems is largely dependent on the application data, this thesis analyzes the dependency between the application data and the simulated low-level effects of fault injection attacks on the software. Combining the application data with software properties helps in detecting low-level fault injections in certain types of software systems with higher precision and accuracy. Finally, the thesis also proposes the use of machine learning to detect system-level threats created by low-level software fault injections.

2.6 Summary

The topics covered in this chapter have given the necessary background information to understand fault injection attacks on software systems and the different types of software fault injection tools. The related work and state of the art methods presented in the chapter clearly demonstrate the need for detection and classification of low-level fault injections in software systems using supervised machine learning and the challenge of creating a system-specific labelled dataset. Chapter 3 will discuss the proposed framework for utilizing supervised machine learning and creation a software system-specific labelled dataset using an instruction-level software fault injection tool.

Chapter 3

Proposed Framework

This chapter describes the proposed framework for detecting and classifying low-level fault injections in software systems using supervised machine learning. The chapter begins with an overview of the proposed framework and subsequently explains each stage of the framework in detail. It explains the software system-specific labelled dataset generation and recommends specific data analysis to be performed on the features of the generated dataset in order to ensure the dataset quality. Finally, the chapter explains the steps involved in training and evaluating the supervised machine learning model.

3.1 Framework Overview

The section explains the overview of the proposed framework consisting of four stages: labelled dataset generation, data preprocessing, data analysis and feature selection, and supervised machine learning to generate a model capable of detecting low-level fault injections in the live environment of software systems. The proposed framework shown in Fig. 3.1 assumes that the software fault injection campaign is performed at an instruction level on the software executables to simulate the effects of fault injection attacks directly on the software [6]. An instruction-level fault injection campaign enables the machine learning model generated by the framework to detect low-level fault injections in the software. Various software fault injection tools operate at different levels on the software. The two main levels of abstraction are high-level and low-level. As explained in Section 2.2, high-level tools simulate the effects of fault injection attacks by modifying the software source code. This makes the simulations specific to a programming language and can only be performed at compile-time.

On the other hand, low-level software fault injection tools operate either at the instruction-level [12] or the microarchitectural-level [30]. Although microarchitectural-level tools replicate the microprocessor fault injections more closely, simulations are difficult as prior knowledge about the microprocessor architecture is required, which may be complex. Microarchitectural-level tools are also limited by the processor emulation platform used and the number of microprocessor architectures supported by the emulation platform. Thus, an instruction-level software fault injection tool is utilized in the proposed framework for dataset generation to simulate the fault injection attack effects at a low-level on the software. Instruction-level tools only require the instruction-set architecture for the software executable and make the simulations easier to implement.

The labelled dataset generation stage involves subjecting the software system to an instruction-level software fault injection campaign. The labelled dataset records the execution output of the executables created in the software fault injection campaign. The dataset generation stage is generic because the use of an instruction-level software fault injection tool does not require a prior knowledge about the source code or other software-specific features. The target label in the dataset is the instruction-level software fault model used for the fault injection in the executable, whereas the features are generic run-time software properties such as output logs, error logs, exit code, and time out [6]. The use of such generic run-time software features makes the proposed supervised machine learning-based framework applicable to a wide range of software systems. The labelled dataset may contain unclean and erroneous data instances. It may also contain features that cannot be used directly in the machine learning stage. The data preprocessing stage generates the final dataset by cleaning the labelled dataset and converting those features into a format suitable for machine learning. For instance, logs that are recorded in the software fault injection

campaign will be in textual format and must be converted into a machine learning suitable format. The output of the data preprocessing stage is the final dataset.

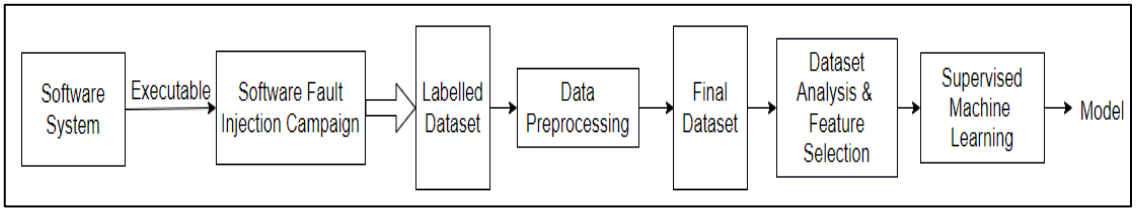

Figure 3.1: Proposed Supervised Machine Learning-Based Framework.

The third stage of the proposed framework is dataset analysis and feature selection. This stage involves analyzing the dataset features using statistical methods and selecting appropriate features for the machine learning stage. This stage ensures that the features included in the machine learning stage are suitable predictors of the fault type. The statistical methods used depend on the type of feature. Feature selection is performed based on the results obtained from the data analysis. This stage gives a preliminary indication of whether or not application data will be useful in the machine learning stage by measuring the prediction strength of each feature with respect to the target label.

The last stage of the proposed framework is supervised machine learning. This stage involves training and evaluating a supervised machine learning model using the selected features to detect low-level fault injections in software systems and classify the fault type. Multiple iterations of labelled dataset generation using different combinations of fault injection control block parameters should be performed depending on the results obtained from the successive data analysis and supervised machine learning stages. This enables selecting the optimal dataset to train a machine learning model providing the best prediction performance.

Fig. 3.2 shows how the supervised machine learning model can be used in the live environment of a software system to detect low-level effects of fault injection attacks. As explained in Sections 1.3 and 2.2, the NOP, FLP, and JMP predicted labels are the low-level attack effects simulated using the instruction-level software fault models. The NONE label indicates prediction of no software fault injections. Unlike other methods proposed in the literature, which perform low-level software fault injections and monitor low-level features such as assembly instruction execution count, register status, and cache status, which are difficult to monitor in the live environment of a software system, the proposed framework utilizes high-level run-time software properties such as output logs and exit codes, which are easily monitored in the live environment of a software system.

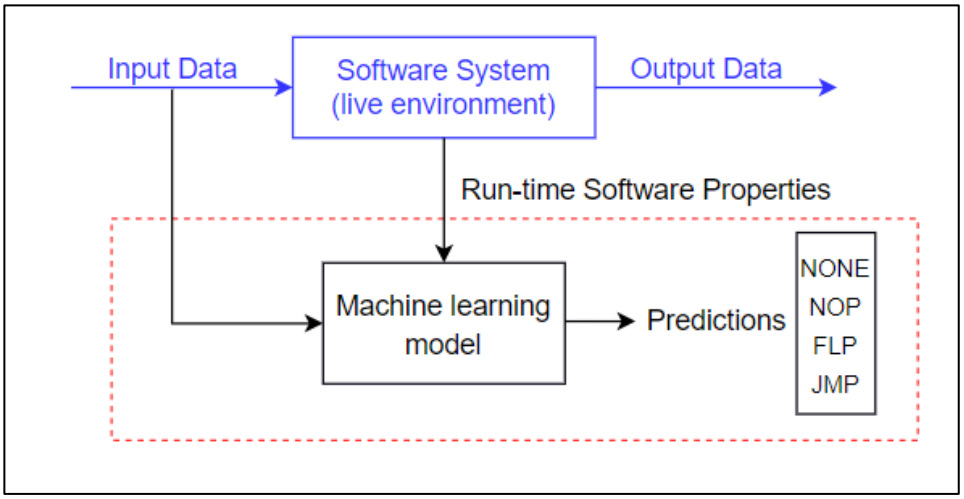

Figure 3.2: Low-level Fault Injection Prediction in Live Environment.

This enables the supervised machine learning model to detect the low-level effects of fault injection attacks in the live environment of a software system. As explained in Section 2.2, the instruction-level fault injected into the executable by the software fault injection tool is a simulation of the low-level effects of fault injection attacks on software systems [6]. The examples shown in Fig. 1.1 and Fig. 1.2 demonstrated that instruction-level software fault

injections can be used to replicate low-level effects of fault injection attacks such as instruction skipping and data corruption [6]. This implies that the supervised machine learning model capable of detecting low-level software fault injections can be utilized in the live environment of a software system to detect the low-level attack effects. In the following sections, each stage of the proposed framework is explained in detail.

3.2 Labelled Dataset Generation and Software Fault Injection Campaign

This section describes how the labelled dataset generation stage of the framework using a dynamic instruction-level software fault injection tool can be utilized to generate a software system-specific labelled dataset for supervised machine learning, as well as the type of features included in the dataset.

Fig. 3.3 shows a software system being subjected to the instruction-level software fault injection campaign. The software fault injection tool simulates fault injection attacks on the executable and generates faulty executables using the specified fault models. The faulty executables are executed using the application input data and the execution output is recorded to generate the labelled dataset. The size of the dataset generated is dependent on the size of the input data used in the software fault injection process and the number of instruction-level faults to be simulated on the software executable. The range of input data used should be selected carefully according to the software system under consideration. Since the hypothesis is that the behavior of certain software systems under fault injection attacks is partially dependent on the application input data, a wide range of input data should be utilized in order to thoroughly test every decision-making block of the software, which may have different behavior for various input data range. This implies that the number of datapoints in the generated dataset can be extremely huge for software systems with a large number of such decision-making blocks, and the size of the dataset needs to be controlled.

The input data utilized in the software fault injection campaign should ideally cover the entire range of inputs expected when the software system is running in a live environment. For an autonomous vehicle, this data may include the entire range of obstacle detection, outdoor temperature, and outdoor light intensity readings that are expected to be encountered in the live environment.

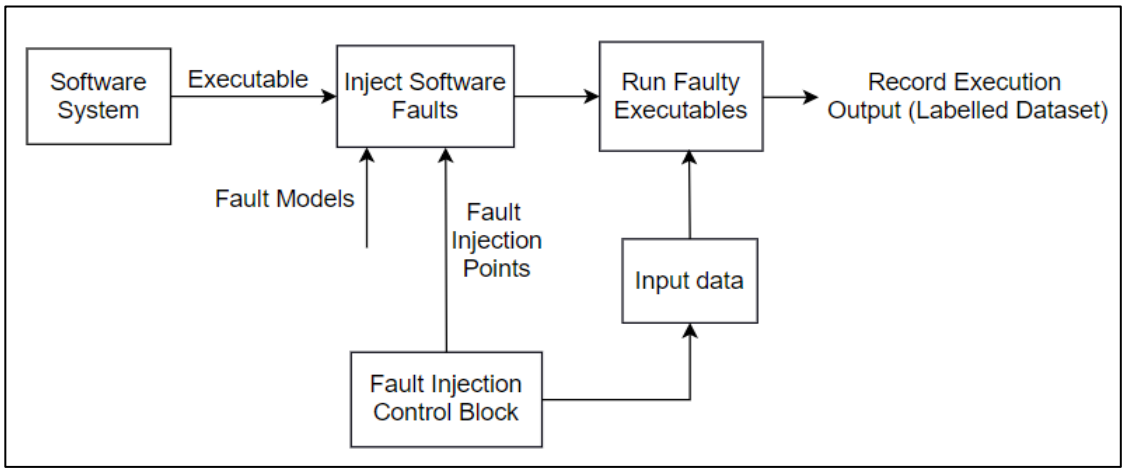

Figure 3.3: Labelled Dataset Generation Stage of Framework.

The fault injection control block is used to control the size of the generated dataset by the software fault injection process. There are two parameters for the fault injection control block. Modifying these two parameters changes the software fault injection points and the application data used in the software fault injection process, which ultimately controls the size of the generated dataset. Along with the execution of faulty executables, the software fault injection tool is also modified to run the software executable with no faults and record the execution output. This is done in order to record the run-time software behavior under normal circumstances when there is no fault injection attack affecting the system.

Eventually, the recorded output of the software fault injection campaign will consist of datapoints generated using both the faulty and normal executables. This collection of recorded datapoints constitutes the labelled dataset. Fig. 3.3 shows the output of the software

fault injection campaign, which is the labelled dataset. The labelled dataset consists of two types of features: generic run-time software features and system-specific features. The run-time software features are recorded dynamically when the executables are run in the software fault injection campaign. These high-level software features such as the output logs and the exit code are generic and are applicable to most of the software systems. The values of the run-time software features provide valuable information indicating the effect of a fault injection attack on the software and also aid in classifying the type of low-level fault injected into the software. For instance, the software timeout being true may indicate the presence of a fault injection, whereas the value of the exit code will help in classifying the type of injected software fault.

The software system-specific features in the dataset include the input data that was used to run the executables during the software fault injection campaign. These system-specific features recorded in the software fault injection campaign should include the various application input data which can affect the software behavior, ultimately affecting the entire system. The behavior of faulty software executables using different fault models at the same injection point varies with application input data. This helps classify software fault injection for systems whose behavior varies significantly for different values of the application input. Apart from the features, each datapoint is labelled with the fault type that corresponds to the fault model used in the software fault injection. The number of distinct labels in the dataset will be equal to one more than the number of fault models used in the software fault injection campaign. This labelled dataset enables the use of supervised machine learning techniques to detect and classify low-level fault injections in software systems with a higher accuracy. The use of instruction-level software fault injection makes the dataset generation stage of the framework generic and applicable to a wide variety of software systems.

3.3 Data Preprocessing

This section explains the data preprocessing stage required to generate the final dataset before the dataset analysis and feature selection stage. Data preprocessing is the next stage in the proposed framework which involves visually inspecting the labelled dataset generated by the software fault injection campaign and deciding the necessary data cleaning techniques to be applied on the dataset before the data analysis and feature selection stage. The dataset needs to be cleaned to ensure there are no invalid and erroneous data instances while analyzing the dataset in order to select features for the machine learning stage. Performing data analysis and feature selection on a dataset containing invalid data instances may lead to an incorrect model building in the machine learning stage.

The data cleaning techniques may include removing duplicates, validating the data type of each feature, handling missing and null values, checking for outliers and extreme values, and other techniques required to clean the labelled dataset. The labelled dataset may include categorical, numerical, textual, and other types of data. The data preprocessing stage is responsible for converting all those features into machine learning-suitable formats. This requires techniques such as natural language processing and unsupervised machine learning to process the features of the labelled dataset and render them in a machine learning suitable format. Machine learning models used in this stage should be thoroughly evaluated before the structured features are used for training supervised machine learning models. For instance, unsupervised clustering models utilized in the data structing stage should be evaluated using metrics such as homogeneity score, adjusted rand index, or silhouette score. The evaluation of unsupervised machine learning models used in this stage may require generating ground truth labels, which necessitates prior knowledge about the software system functioning and behavior. For instance, while converting textual features to categorical labels, knowledge about the expected textual features for a particular application

input value may be required. The data preprocessing stage generates the final dataset to be used for data analysis and feature selection.

3.4 Dataset Analysis and Feature Selection

The next stage in the proposed framework is analyzing the final dataset using data analysis techniques and accordingly select features that are good predictors of the fault type to be used for supervised machine learning. This analysis can be performed using statistical association measures such as correlation coefficients and data visualization techniques such as scatter plots and heat maps. The dataset is split into train and test data before the data analysis and feature selection stage. To ensure that no knowledge of the train data leaks into the test data, all data analysis and pre-processing techniques are exclusively applied to the train data at first. This ensures that the supervised machine learning model is evaluated on completely unseen data.

The type of association measure used is dependent on the type of the two variables under consideration. Since the target label will always be categorical, the association measure used will depend on whether the feature being analyzed is numerical or categorical. The association measures used for categorical features will also depend on whether the feature is nominal or ordinal. The values of the association measures for the features indicate their predictive strength with respect to the fault type, which helps in selecting the final features to be used for the supervised machine learning stage. For any association measure used, the acceptable value for the feature to be included in the supervised machine learning stage should be determined by considering the software system in question. Baseline values of the association measures for each feature should be recorded during the first iteration of the dataset generation stage in order to adjudge the values obtained during successive iterations. The data analysis performed in this stage provides a preliminary indication of the

predictive strength of the application data recorded in the software fault injection campaign with respect to the fault type.

3.5 Supervised Machine Learning - Training & Evaluation

This stage involves training machine learning models on the final dataset to predict and classify the injected fault using a supervised algorithm. This stage is divided into two steps. The first step includes hyperparameter optimization and training the machine learning model. Hyperparameter optimization involves tuning the set of parameters used to initialize the supervised machine learning model prior to training. Hyperparameter optimization can be performed using techniques such as randomized search and grid search. Randomized search trains various machine learning models with randomly selected combinations of hyperparameters and selects the hyperparameter combination that gives the best score as the output. Grid search trains machine learning models using every possible combination of the available set of hyperparameters and selects the combination giving the best score as the output. If the range of hyperparameter values is small, randomized search should be preferred over grid search because grid search is more time-consuming due to its exhaustive testing of every possible combination. The range of hyperparameter values should be selected so that the model does not overfit and performs well on both train and test data. The scoring strategy used to select the best set of hyperparameters may include classification metrics such as accuracy, precision, recall, f1-score, or any other user-defined evaluation metric.

Subsequently, a machine learning model is trained using the train data, a supervised algorithm and the corresponding optimized hyperparameters. The next step in this stage involves evaluating the trained supervised model using the above-mentioned classification metrics. The evaluation metric should be chosen with the objective of correctly classifying

as many fault injection instances as possible while minimizing the number of incorrectly classified instances. Nested Cross Validation (Nested-CV) should be used to ensure the consistency and statistical significance of the obtained evaluation metrics. Nested-CV uses an outer loop to perform K-fold Cross Validation (K-fold CV) and an inner loop to perform hyperparameter optimization. As a result, hyperparameter optimization is performed for different combinations of train and test data. Nested-CV prevents overfitting and underfitting by evaluating the model with the optimal hyperparameters on multiple combinations of train and test data.

Feature importance should be used to measure the usefulness of individual features in making the predictions. The feature importance scores indicate the contribution of application input data in the predictions for software systems where the application input data was included as features in the machine learning stage. The output of this stage is a supervised machine learning model that can predict low-level fault injections in software systems. The proposed framework combines detection and classification of the fault type into a single task because the instances without any injected faults are labelled as NONE.

Fig. 3.2 shows the detection and classification of low-level fault injections in the live environment of a software system. The run-time software properties and the input data are monitored and used by the machine learning model to generate the predictions. These run-time software properties correspond to the software properties captured during the software fault injection campaign in order to generate the dataset, such as the output and error logs, exit code, and time out. The use of such high-level software properties as features for the machine learning model make it applicable to a wide range of software systems. Further, the easy monitoring of these software features compared to low-level features such as assembly instruction execution count, register status, and cache status makes the model generated by the proposed framework usable in the live environment of a software system. The

classification of low-level fault injections will also assist in determining the necessary corrective measures to safeguard the system against any adverse effects. For instance, fault injection attacks can cause control flow modifications in the software execution because of instruction skips, which can be replicated using the NOP flip model. A correct classification in the live environment of a software system will help in deciding the necessary countermeasures against the control flow modifications.

3.6 Summary

This chapter has presented the proposed supervised machine learning-based framework consisting of labelled dataset generation for detecting and classifying low-level fault injections in software systems using an instruction-level dynamic software fault injection tool. The chapter also describes how the machine learning model generated by the framework can be used in the live environment of a software system to detect the low-level effects of fault injection attacks. Chapter 4 will discuss the implementation of the proposed framework using the chaos duck instruction-level software fault injection tool and the case studies performed to demonstrate and evaluate the framework.

Chapter 4

Implementation & Case Studies

This chapter begins by explaining the tools and libraries used in the implementation of the proposed framework. The chapter then explains the modifications made to the chaos duck tool for implementing the dataset generation stage of the framework. Subsequently, the chapter explains the data analysis and feature selection performed before the machine learning stage. The chapter concludes by demonstrating the implementation of the supervised machine learning stage and describing the safety-critical VerifyPIN software and the Nova Smart Home IoT software used in the case studies.

4.1 Tools

The section explains the working of the Chaos Duck software fault injection tool [12] used to conduct the software fault injection campaign on the software executable. The section also describes the machine learning libraries and data visualization tools such as Scikit-learn, Gensim, and Seaborn used in the implementation of the proposed supervised machine learning-based framework.

4.1.1 Chaos Duck Software Fault Injection Tool

Chaos duck is chosen as the instruction-level software fault injection tool for implementation of the labelled dataset generation stage due to its support for multiple commonly used instruction set architectures. Unlike other instruction-level tools that support limited software fault models, chaos duck supports a wide range of fault models. These models include bit-level and byte-level models such as JMP, FLP, NOP, Zero One Byte (Z1B), and word-level models such as Zero One Word (Z1W). In comparison, other instruction-level

software fault injection tools such as Lazart [34] and EFS [37] only support limited fault models and a single instruction set architecture. The currently supported instruction set architectures by chaos duck include x86_64, x86_32, and ARM, and support for additional architectures can be easily added using a third-party library or plugin. The inbuilt ability of the tool to execute and record the output of the faulty executables also makes it a suitable candidate. Fig. 4.1 shows the working of the chaos duck tool. The chaos duck tool accepts a software executable as an input. The tool is programmed in python and chaosduck.py is the python script file that needs to be executed to begin with the software fault injection process.

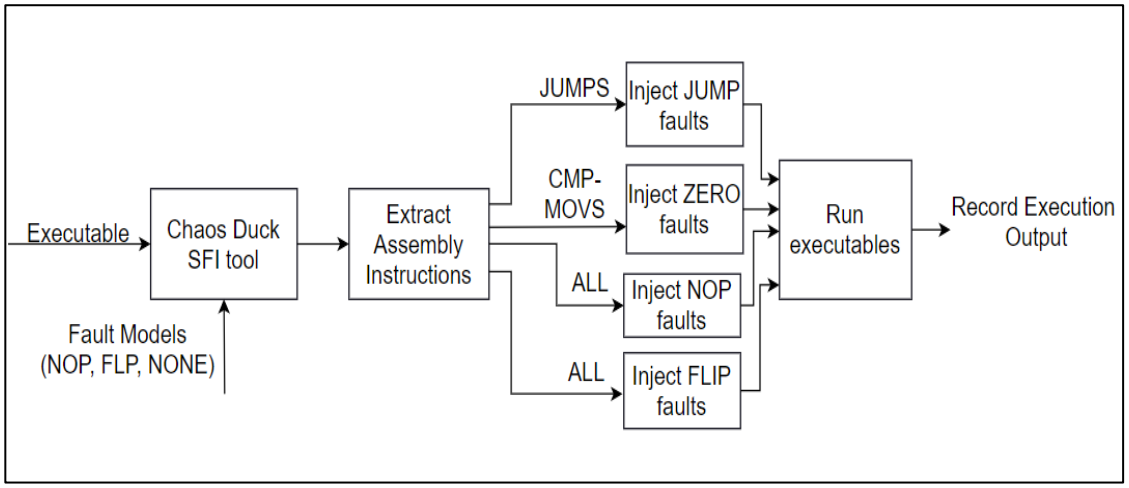

Figure 4.1: Chaos Duck Software Fault Injection tool.

The python file should be executed with the name and architecture of the software executable as command line arguments. According to the type of architecture, the tool parses the software executable and extracts all the assembly instructions from the executable. Chaos Duck uses the Capstone disassembling library [51] to extract the assembly instructions. The Capstone library supports both x86 and ARM software executables. The extracted instructions are categorized into ALL, JUMPS and CMP-MOVS. The ALL category contains all the assembly instructions extracted from the software executable. The JUMPS

category contains only the jump instructions present in the software executable. This includes conditional as well as unconditional jump instructions. The CMP-MOVS category contains the compare and move instructions present in the software executable. Different instruction categories are used to inject faults into the software executable using various fault models.

Once the assembly instructions are extracted and categorized, the chaos duck tool begins with the software fault injection process. The tool currently supports four different fault models for software fault injection. The Jump fault model injects faults into software executables by modifying the conditional and unconditional jump instructions extracted into the JUMPS category [6]. The Jump fault model can be used to replicate the effects of fault injection attacks that cause conditional checks to return an incorrect value. The Zero fault model is a byte-level fault model that injects faults into software executables by changing a single byte of the instructions extracted into the CMP-MOVS category to zero. The Zero fault model can be used to replicate the effects of fault injection attacks that cause instruction skipping and program corruption. The NOP fault model is another byte-level fault model that injects faults into software executables by replacing instructions in the ALL category with an NOP instruction. The NOP instruction is an assembly command that does nothing. The NOP fault model can be used to replicate the effects of fault injection attacks that cause instruction skipping and control flow modifications.

Finally, the bit-level Flip fault model is used to flip bits of instructions in the ALL category from 0 to 1 or 1 to 0. The number and position of bits to be flipped in an instruction byte can be controlled in the chaos duck tool by setting the variable significance bit. For instance, if the variable significance bit is set to 6, the 6 most significant bits of each instruction byte will be flipped. Since this is a bit-level fault model and is applied to all

instructions, it injects significantly greater number of faults into any software executable compared to all other fault models. The effects of flip faults depend on the instruction and the number of significant bits being modified. Thus, the Flip fault model can be used to replicate a variety of fault injection attack effects by injecting low-level faults into the software. The chaos duck tool uses the Software Implemented Fault Injection Tool (swifi-tool) [52] to inject faults into the software executable using the specified fault model and generates a faulty executable for each injected fault.

Each faulty executable is subsequently executed using the application input data required by the software executable. The chaos duck tool must be configured with the source of application input data required to run the faulty executables. The tool executes each faulty executable for all application input values. The execution output of each faulty executable is recorded in a Comma-Separated Values (CSV) file, which contains the name of the faulty executable that was run, the application input data, and the output logs that it generated. The output logs generated by running the faulty executables must be analyzed to determine the vulnerability of a software executable to fault injection attacks. This analysis can be performed manually by a domain expert or can be automated by utilizing data analysis and machine learning techniques.

4.1.2 Scikit-learn

Scikit-learn, also known as sklearn, is an open-source machine learning library supporting the python programming language. It features most of the machine learning algorithms used for classification, regression and unsupervised clustering. Scikit-learn is designed to support other commonly used data analysis and processing libraries, such as NumPy, SciPy, Pandas, and many others. Scikit-learn can be used for training, evaluating and comparing machine learning algorithms. The scikit-learn library was used in this thesis due to the ease with which

new machine learning models can be instantiated and the statistical tools provided to analyze the trained models. In addition, scikit-learn also offers RandomizedSearchCV and GridSearchCV for optimizing the hyperparameters of the trained model. Scikit-learn combines hyperparameter optimization-based training and K-fold cross validation K-fold CV based evaluation of a supervised machine learning model into a single step called Nested-CV. This makes it very convenient to test multiple models with different hyperparameters and combinations of train and test data in order to select the best performing model.

Scikit-learn also provides numerous data exploration and feature selection functions which are useful in the dataset analysis stage. For instance, the SelectKBest feature selection function can be used to measure the mutual information between any two features of the dataset. Supervised machine learning algorithms such as K-NN and Random Forest are implemented in scikit-learn with an inbuilt support for utilizing multiple cores for the computation. This feature is most useful during the resource-intensive Nested-CV process. Other features of scikit-learn such as balanced partitioning during train test split and dummy classifier to create a benchmark model for evaluation make it a suitable library to be used for machine learning in this thesis.

4.1.3 Gensim

Gensim is a python-based open-source statistical machine learning library used for unsupervised modeling and natural language processing tasks. The gensim library is used in this thesis because it provides APIs for common text processing such as removing punctuations, tags, multiple whitespaces, and numbers. Apart from that, it also provides some advanced text processing APIs such as stemming the individual words in the text and

stripping them short to reduce the number of features during machine learning and reduce the noise in the data.

4.1.4 Seaborn

Seaborn is a python-based data visualization library that extends and improves the functionality of the matplotlib library. The seaborn library offers high-level functions for creating visually appealing and informative statistical graphs. Seaborn is compatible with the commonly used data manipulation libraries in python, such as Pandas. It generates graphs from commonly used data structures in python such as numpy arrays and pandas dataframes. The declarative nature of the APIs offered by seaborn makes it easy to use. Seaborn is used as the data visualization tool in this thesis to explore and analyze the generated datasets. For instance, the barplot function of the seaborn library is used to plot and compare the evaluation metrics of the supervised machine learning models trained on the datasets, whereas the lineplot function is used to plot the multiple evaluation metrics obtained during each fold in the Nested-CV process.

4.2 Implementation Details

This section explains the implementation of each stage of the proposed framework in detail, along with the utilized tools and libraries. Fig. 4.2 and Fig. 4.3 demonstrate the implementation of labelled dataset generation and supervised machine learning for the case studies, respectively.

Fig. 4.2 shows a software system with its system-specific input and output data. For example, the input for an autonomous vehicle software may include obstacle detection, outdoor temperature, and outdoor light intensity, and based on this input, the software system of the vehicle may control output devices such as the motors, air conditioner and lights. The subsequent sections describe each stage of the implementation in detail.

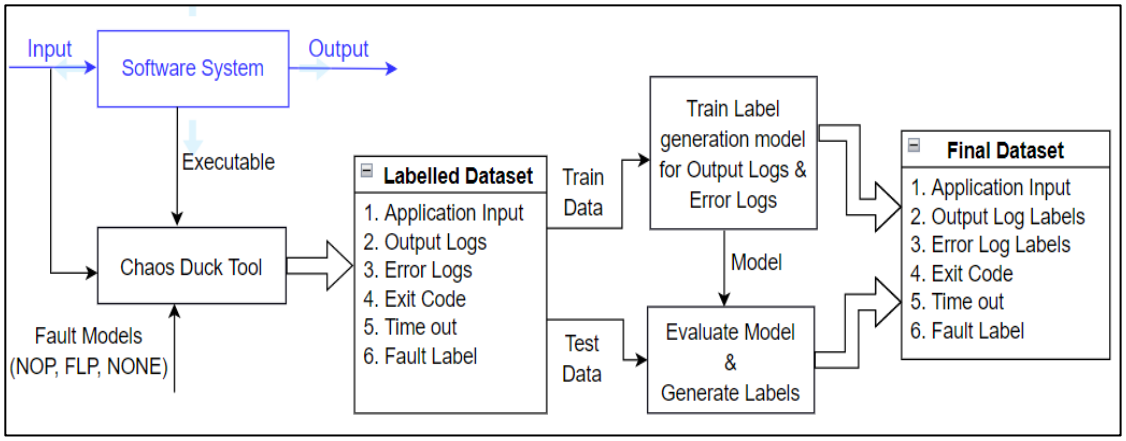

Figure 4.2: Implementation of the Labelled Dataset Generation Stage of Framework.

4.2.1 Labelled Dataset Generation

The use of the modified chaos duck software fault injection tool to implement the software system-specific labelled dataset generation is illustrated in Fig. 4.2. In addition to running the faulty software executables, the tool is modified to run the software executable without any injected faults using all the application input data that was used to run the faulty executables. This execution output is recorded with the label NONE in the CSV file generated by the chaos duck tool. The chaos duck tool is also modified to record the name of the faulty executable that was run, application input data, output log, error log, exit code, time out, and the fault type that was injected into the software executable. These run-time software properties constitute the features of the labelled dataset. Additional run-time features can be captured by modifying the software fault injection tool. Along with the software executable name and architecture, the tool is modified to receive a list of software fault models and the source of application input data to run the faulty executables from the command line. The chaos duck tool will only use the software fault models specified on the command line in the software fault injection process. This makes the tool more flexible, as multiple instances of the python script can be executed simultaneously using different fault

models, provided the machine performing software fault injection has sufficient resources to support it.

Next, modifications are made to the chaos duck tool to control the size of the generated dataset for different software systems. This constitutes the fault injection control block of the dataset generation stage. As explained in Section 3.2, the size of the dataset can be controlled either by modifying the size of the application data used or the number of faults simulated on the executable in the software fault injection campaign. Thus, multiple versions of the *run_faulty_executables* function are created in order to implement the fault injection control block. Table 4.1 lists the functions created to implement the control block.

Table 4.1. Functions to Control Size of Generated Dataset.

Input	**Fixed No. of Randomly Selected Executables**	**All Faulty Executables**	**Predetermined Faulty Executables**
All input	*run_random_faulty_executables*	*run_faulty_executables*	*run_fixed_faulty_executables*
Randomly selected input	-	*run_all_faulty_executables_random_input*	-

The plain version of the function is *run_faulty_executables*, which executes all the faulty executables created for a particular software fault model using every instance of application input data. For instance, if there are 13,000 faulty executables and 1,000 application input data instances, the tool will generate a total of 13,000,000 instances in the dataset. This demonstrates that the function is not feasible when both the number of injected faults and the size of application input data are large.

The *run_random_faulty_executables* function is created to run a fixed number of randomly selected executables for each software fault model utilizing each application input data instance. The number of randomly selected faulty executables for each input instance is regulated using the *batchsize* parameter. If *batchsize = 1*, this function will generate a dataset with the number of datapoints equal to the number of application input data instances utilized in the software fault injection campaign. An extra argument is added to this function called *duplicate*. If *duplicate* = False, the function removes a faulty executable from the list of available faulty executables after it has been executed using an application input data instance. This parameter prevents recording the behavior of a faulty executable for different application input data and should not be used to generate datasets for software systems whose behavior varies significantly for different range of input values. Next, the *run_all_faulty_executables_random_input* function is created to run all the faulty executables for a particular software fault model using randomly selected application input data instances. This function should be used to record the software behavior for a large number of software fault injection points but a limited number of application input data instances.

Finally, the *run_fixed_faulty_executables* function is created to run a fixed number of predetermined faulty executables for each software fault model. This function should be used when the size of the application input data used in the software fault injection campaign is large and the dataset is to be created for a fixed number of predetermined fault injection points in the software executable. This function records the software behavior for the same software fault and a large number of distinct application input data instances. Additionally, multiprocessing is added to all the execution functions created in the chaos duck tool. This drastically reduces the dataset generation time, especially for functions that deal with a large

number of application input data instances by facilitating the simultaneous execution of the same faulty executable for multiple batches of input data.

4.2.2 Data Preprocessing

The data preprocessing step converts the output and error logs into categorical labels that are suitable to be used in machine learning algorithms. Fig. 4.2 illustrates the label generation process for the output and error logs using a clustering algorithm. The labelled dataset is split into train and test data at a ratio of 60% and 40%, respectively. Before applying the clustering algorithm to generate the log labels, some preprocessing needs to be performed on the logs. The logs are pre-processed using the Gensim library for natural language processing [53]. The logs are stripped of all the punctuation, tags, multiple whitespaces, numbers, and stopwords. As the semantic meaning of the logs is of interest, stemming is applied to convert all words to their root forms.

Once the data is cleaned, the next step is to train a clustering model using the train data for partitioning the logs and generating the labels. For small to medium sized logs, the unsupervised k-means clustering algorithm can be utilized if the approximate number of clusters “k” is known beforehand. For software systems where the size of logs is relatively large, hierarchical clustering algorithms that do not require specifying the number of clusters can be utilized. For instance, Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is an unsupervised clustering algorithm that does not require specifying the number of clusters to be formed but only requires specifying the minimum number of datapoints for a particular neighborhood to be classified as a separate cluster. Gaussian mixture model (GMM) is another type of unsupervised clustering algorithm which uses the probability distribution of the dataset to generate clusters.

Separate clustering models are trained for the output and error logs. The labels generated for the output and error logs are used as features while training the supervised machine learning model in the subsequent stages. This makes it necessary to ensure that the labels generated in this stage are reliable enough to be used in the subsequent stages. To this end, the unsupervised clustering machine learning models are used to generate labels for the test data and are evaluated using clustering metrics such as homogeneity score, adjusted rand index, and silhouette score. The homogeneity score indicates how well the data has been partitioned using clustering, with 0 indicating poor partitioning and 1 indicating optimal partitioning. The adjusted rand index is a measure of the extent of similarity between any two data clusters, with -1 representing the most similarity and 1 indicating the least similarity between any two clusters. The ground truth is required to calculate the above two parameters. On the contrary, the silhouette score is a parameter that does not require ground truth and makes use of the clustering model and input data. Once the evaluation metrics for the test data are deemed acceptable, the train and test data are merged to form the final dataset, which is used to train the supervised machine learning model for low-level fault injection detection in the subsequent stages.

4.2.3 Dataset Analysis and Feature Selection

The final dataset is analyzed to ensure that the features to be used for supervised machine learning are good predictors of the fault type. Fig. 4.3 illustrates the implementation of the dataset analysis and feature selection stage. The injected fault type label in the dataset is a categorical variable. The dataset analysis is performed using the stats package of the SciPy scientific computing library [54] and the feature_selection package of the Scikit-learn machine learning library [55]. The Seaborn data visualization library [56] is utilized to visualize the results of dataset analysis and select the appropriate features.

The categorical features in the dataset are analyzed using the Cramer's V coefficient. The Cramer's V coefficient measures the association between two nominal variables and ranges between 0 and 1. A value of 0 indicates no association whereas a value of 1 indicates complete association. The numerical features in the dataset are analyzed using the Kruskal-Wallis test. The Kruskal-Wallis test is predicated on the null hypothesis, which states that the two variables have no relationship. The p-value generated by the test indicates whether or not the null hypothesis can be rejected. Lower the p-value, higher the probability that the feature is a good predictor of the fault type. For instance, a p-value of 0.05 indicates that the null value can be rejected and the two variables have a statistical relation with a 95% confidence level. The mutual information score between each feature and the fault type is also measured. This measure is applicable to both numerical and categorical features and ranges between 0 and 1. It measures the reduction in uncertainty for the target label, given a known value of the feature.

4.2.4 Supervised Machine Learning - Training & Evaluation

Once the final dataset is generated, the next stage to implement in the proposed framework is training the supervised machine learning models. The machine learning models are trained and evaluated using the Scikit-learn machine learning library [55]. Fig. 4.3 illustrates the steps involved in training a supervised machine learning model on the selected features from the final dataset.

The goal of using supervised machine learning is to correctly classify the majority of actual low-level software fault injections, the true positives, while minimizing the number of incorrectly classified actual low-level software fault injections, the false negatives. Therefore, recall is considered as the most important metric in the case studies. The precision is reported as the secondary metric because it indicates how well the model was able to

minimize the number of instances that were incorrectly classified as low-level software fault injections, which are the false positives. Accuracy is not used as an evaluation parameter because it does not take into account the type of wrong prediction and only provides a gross value of the model capability. Recall, precision, and f1-score are reported as the evaluation parameters for each supervised machine learning model trained in the two case studies. However, recall is given the highest priority in selecting the best performing machine learning model because of the goal of minimizing the number of false negatives.

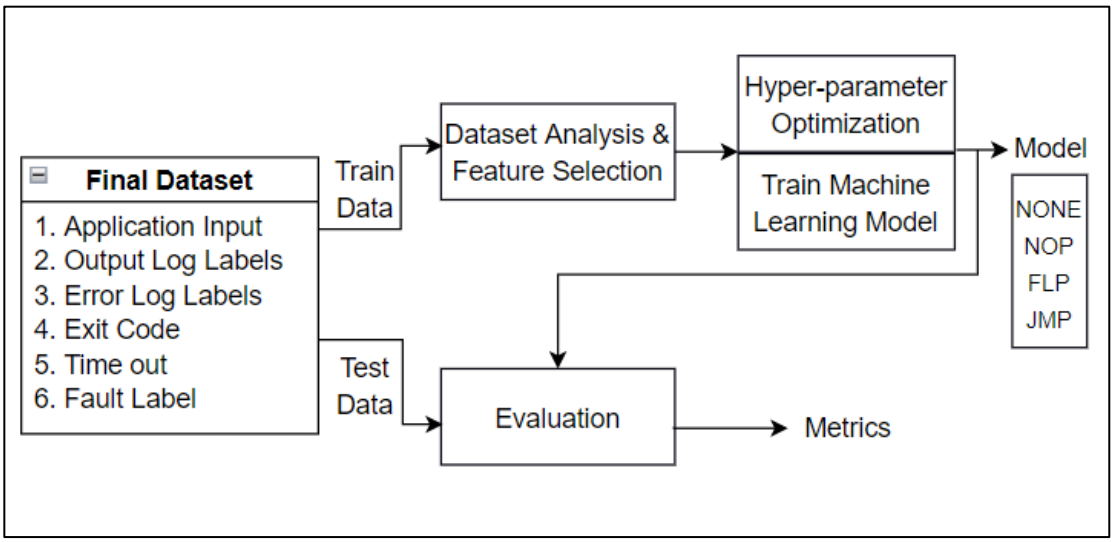

Figure 4.3: Implementation of the Supervised Machine Learning Stage of Framework.

The Nested-CV is performed to demonstrate the consistency of the evaluation results using n_splits = 10 for the outer loop which performs the stratified k-fold cross validation. This implies that the machine learning model is trained and evaluated using 10 distinct and stratified combinations of train and test data. The inner loop used for randomized hyperparameter optimization is executed using n_splits = 5 and n_iter = 5. This implies that 25 distinct hyperparameter combinations are tested for each combination of train and test data.

4.2.5 System-Level Threat Detection

This section explains the use of supervised machine learning on the generated low-level software fault injection datasets to detect system-level threats. The machine learning model can subsequently be utilized in the live environment of the software system to predict if a fault injection attack resulted in a system-level threat. Fig. 4.4 shows the training of a supervised machine learning Model (M1) to predict the ideal output log label for various application inputs. The model is trained with the output log labels as the prediction target and the corresponding application input data as the features. The model M1 is trained using only the instances labelled as NONE in the final dataset created by the software fault injection campaign. The fault injected instances in the final dataset cannot be used to train this model because they do not represent the ideal output log label for each input.

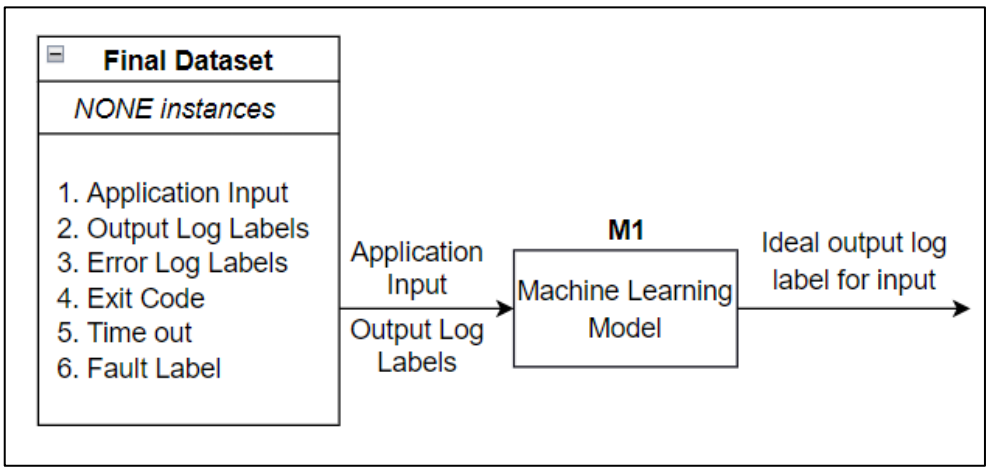

Figure 4.4: Training Machine Learning Model for System-Level Threat Detection.

Fig. 4.5 shows the use of model M1 during the software run-time in a live environment to detect system-level threats. The input to model M1 would be the instances detected as low-level fault injections in the live environment as explained in Section 3.5. These detected instances are low-level effects of fault injection attacks on software systems. The application input from the detected injection instances would be fed to the model M1 to generate the

ideal output log label. Identical ideal and actual output log labels indicate that the software behavior in the presence of a fault injection attack was similar to the normal software behavior. This indicates that the fault injection did not generate a system-level threat.

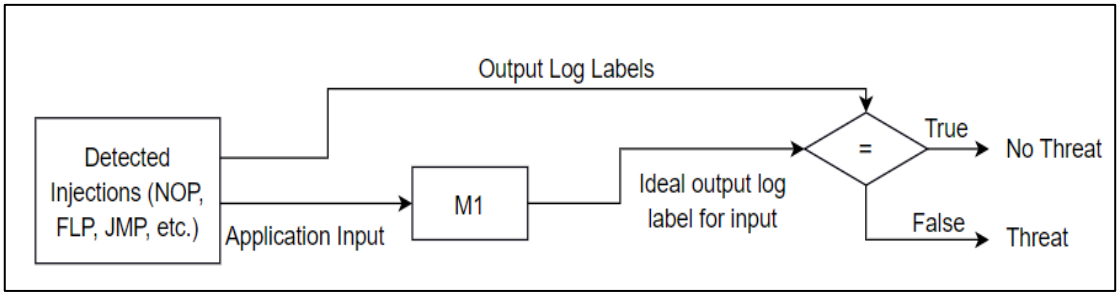

Figure 4.5: System-level Threat Detection in a Live Environment.

In contrast, non-identical ideal and actual output log labels indicate that the software behavior in the presence of a fault injection attack was different than the normal software behavior. This indicates that the fault injection generated a system-level threat. In this manner, the final dataset generated by the proposed framework can also be used to train a supervised model which can predict whether or not the detected low-level effect of fault injection attack resulted in a system-level threat.

4.3 Case Studies

This section explains the case studies conducted on two software systems to demonstrate the implementation and evaluate the feasibility of the proposed framework.

4.3.1 Safety-critical PIN Verification Software

The VerifyPIN algorithm implemented in C from the FISSC dataset [13] is chosen as the safety-critical software for the first case study. The motivation behind choosing this software is to demonstrate that fault injection attacks on this PIN verification software, widely used in several personal and smart digital devices to authenticate the device owner, can result in unauthorized access. The instruction-level effects of such attacks can be detected using the

supervised machine learning model generated by the proposed framework. This benchmark software is hardened using several measures and has been used in the literature for analysis of software robustness against fault injection attacks. The VerifyPIN software compares a 4-digit user PIN to a 4-digit card PIN. The byteArrayCompare function compares each byte of the user-entered PIN to the card PIN and authenticates the user if the two PINs match. The hardened version of VerifyPIN is used in several personal and smart digital devices to authenticate the device owner. A dynamic instruction-level software fault injection tool is used to demonstrate that the hardened VerifyPIN program remains vulnerable to low-level faults caused by fault injection attacks. This can cause control flow alterations, granting unauthorized access to the attackers. Fig. 4.6 shows an example of the hardened VerifyPIN program authenticating an invalid user pin in the presence of a FLP fault in the software executable.

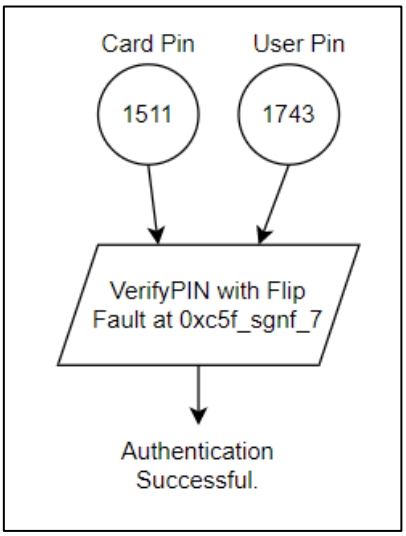

Figure 4.6: FLP fault in VerifyPIN granting unauthorized access.

The card and user PINs are passed to the VerifyPIN software via the command line arguments. The software fault injection for the x86-64 executable of the VerifyPIN software campaign is carried out on a Linux based system. The software fault injection parameters

include the source file of application input data, name and instruction set architecture of the executable, and the list of fault models. Randomly generated combinations of 4-digit card and user PINs are used as input during the software fault injection campaign on the VerifyPIN software executable. The input data consists of 1000 combinations of randomly generated PINs, consisting of same as well as different card and user PINs. The labelled dataset generated for this case study includes six features and one target label. Card pin, user pin, output log, error log, exit code, and time out are the features included in the labelled dataset. Each datapoint is labelled with the corresponding software fault model that was used to generate it. The possible values of the fault type in the case study are the NONE, FLP, and NOP fault models. Two iterations of the labelled dataset generation stage were carried out using *run_fixed_faulty_executables* and *run_random_faulty_executables* functions of the fault injection control block, respectively. The second iteration utilized *run_random_faulty_executables* function with *batchsize*=50 for the faulty executables and all of the input data instances. The dataset analysis and feature selection stage yielded positive results for this dataset, so no further iterations were performed. The labelled dataset contained 602,051 instances, with an approximately equal proportion of NONE, FLP, and NOP labels.

Considering the size of logs generated by the VerifyPIN software, the k-means clustering algorithm is used to generate the labels for the output and error logs in the data preprocessing stage. The k-means machine learning models are trained using the Scikit-learn machine learning library [55]. The k-means models trained with a ratio of 50:50 for train and test data performed equally well with a ratio of 70:30, so the ratio was maintained at 50:50. The number of clusters to be formed from the data (k) is equal to the expected number of categorical labels for the logs. The value of k is set to 4 for training both clustering models. The clustering metrics for the k-means clustering models used to partition and label the

output and error logs are measured using the metrics mentioned in Section 4.2.2 in order to ensure that the generated log cluster labels are reliable. For the VerifyPIN software, it is assumed that a feature with Cramer's V score greater than 0.1 can positively contribute to the prediction of the fault type. Similarly, it is assumed that a feature with mutual information score greater than 0.1 can positively contribute to the prediction of the fault type. The Kruskal-Wallis p-values are not measured for this case study as the dataset does not include any numerical features.

4.3.2 Nova Smart Home IoT software

The second case study is performed on the Nova Smart Home [14], which is an open-source home automation project hosted on GitHub. An IoT software is chosen due to the various application-level input data used by such software, such as temperature, light intensity, and humidity. Unlike other studies in the literature, which mostly consider safety-critical software such as cryptography or PIN verification software, choosing this IoT software demonstrates that fault injection attacks can significantly affect application software and cause the output devices of a system to behave abnormally. The supervised machine learning model generated by the proposed framework can be used to counter these threats. The control daemon for this project is written in C++ and runs on a Raspberry Pi 3 Model B as an ARM software executable. This piece of code controls multiple devices such as a heater, an air conditioner, and lights based on the input it receives from the sensors. For instance, the heater is activated when the temperature falls below a predefined threshold.

The input to the control daemon includes the temperature and light intensity. As the fault injection attacks on IoT software are simulated using a software fault injection tool, the control daemon code is modified to receive temperature and light readings from the command line instead of the physical sensors. This enables the software fault injection tool

to run the software executables by passing the temperature and light readings from the command line. The output logs generated on execution of the control daemon software indicate how the devices are controlled for different input ranges. Since the raspberry pi has limited resources, the code is compiled with the static flag (-static) enabling the software fault injection campaign to be carried out on a Linux-based system. If a Raspberry Pi is unavailable, the software fault injection campaign can be conducted on Linux-based systems using the arm-linux-gnueabi-gcc cross-compiler [57].

Similar to the VerifyPIN software, the execution parameters include the source file of application input data, name and instruction set architecture of the executable, and the list of fault models. As mentioned in Section 3.2, the range of input data chosen for the software fault injection campaign is dependent on the software application. In this case study, temperature readings between -30° C and 100° C, and light intensity readings between -900 lux and 2250 lux are used. Since the hypothesis is that the behavior of the IoT software under simulated fault injection attacks is partially dependent on the application input data, the input values are generated randomly within a wide range in order to test the software thoroughly. Three iterations of the dataset generation stage using *run_fixed_faulty_executables*, *run_random_faulty_executables*, and *run_all_faulty_executables_random_input* functions of the fault injection control block were performed, respectively. The third iteration utilized the *run_all_faulty_executables_random_input* function with *batchsize*=50 for the input data instances and all of the faulty executables. The dataset analysis and feature selection stage yielded positive results for this dataset, so no further iterations were performed. The labelled dataset contained 284,510 instances, with NONE, FLP, and NOP labels in an approximate ratio of 4:2:1.

The labelled dataset for this case study includes six features and one label. Temperature and light intensity are the numerical features, whereas output log, error log,

exit code, and time out are the categorical features included in the labelled dataset. Each datapoint is labelled with the corresponding software fault model that was used to generate it. The possible values of the fault type in the case study are the NONE, FLP, and NOP fault models. Similar to the VerifyPIN case study, the k-means clustering algorithm is used to train the models for output and error log label generation and the clustering metrics for both the models are measured. Since the performance of k-means models using train and test data ratios of 50:50 and 70:30 did not differ significantly, the ratio was maintained at 50:50. The number of clusters to be formed from the data (k) is set to 8 for training both clustering models in this case study. The acceptable values of Cramer's V score, Kruskal-Wallis p-value, and mutual information score are set to 0.1, 0.5, and 0.1, respectively.

4.4 Summary

This chapter has detailed the implementation of the proposed framework by performing case studies on two software systems. The chapter explains the rationale behind the machine learning libraries and data visualization tools utilized in the case studies. The implementation details provide information on the modifications made to the chaos duck software fault injection tool, and how the modified tool can be used to generate a labelled dataset for supervised detection and classification of instruction-level fault injections in software systems. Chapter 5 will present the results obtained from the case studies and analyze them to evaluate the proposed framework.

Chapter 5

Evaluation Results

This chapter presents the results obtained from the case study-based evaluation of the proposed framework. The chapter begins by presenting the results obtained from the data preprocessing and analysis of the software system-specific labelled datasets generated in the case studies using the dataset generation stage of the framework. Subsequently, the chapter presents the results obtained from the training and evaluation of multiple supervised machine learning models on the datasets generated for the two software systems. The results are analyzed to compare the performance of different supervised machine learning algorithms on the two datasets and demonstrate the efficacy of the proposed supervised machine learning-based framework.

5.1 Dataset Preprocessing

The results obtained from the data preprocessing stage of the framework are presented for the two case studies. The results obtained by evaluating the unsupervised clustering machine learning models used in this stage determine whether the structured features are reliable enough to be used for the supervised machine learning stage.

5.1.1 VerifyPIN

The clustering metrics obtained from labelling the output and error logs in the data preprocessing stage for the VerifyPIN software using separate k-means clustering models are shown in Table 5.1. The high values of these clustering metrics indicate that four clusters were sufficient to partition the output and error logs in the labelled dataset generated by the software fault injection campaign. While a score of 1 indicates perfect partitioning, the

values of 0.872, 0.968, and 0.927 for homogeneity score, adjusted rand index, and silhouette score for the output logs indicate near-perfect partitioning. The homogeneity score of 0.872 indicates that each cluster is 87.2% pure, implying that 87.2% of the output log labels in each cluster are identical on average. The adjusted rand index of 0.968 indicates that 96.8% of the output log labels are different between any two clusters on average. The silhouette score of 0.927 also indicates that each cluster created for output log labels is 92.7% pure. However, it is calculated using the input data and clustering model without using ground truth labels. These high values of clustering metrics suggest that the output log and error log labels can be used reliably as features in the supervised machine learning stage.

Table 5.1. VerifyPIN: Clustering metrics for partitioning of output and error logs.

Clustering Model	**Train Time**	**Homogeneity Score**	**Adjusted Rand Index**	**Silhouette Score**
Output Logs	0.379s	0.872	0.968	0.927
Error Logs	0.261s	0.936	0.864	0.825

Table 5.2. Nova Smart Home: Clustering metrics for partitioning of output and error logs.

Clustering Model	**Train Time**	**Homogeneity Score**	**Adjusted Rand Index**	**Silhouette Score**
Output Logs	0.527s	0.992	0.996	0.870
Error Logs	0.623s	0.982	0.874	0.857

5.1.2 Nova Smart Home

The clustering metrics obtained from labelling the output and error logs in the data preprocessing stage for the Nova Smart Home software using separate k-means clustering models are shown in Table 5.2. The high values of these clustering metrics indicate that eight clusters were sufficient to partition the output and error logs in the labelled dataset generated by the software fault injection campaign. The values of 0.992, 0.996, and 0.870 for homogeneity score, adjusted rand index, and silhouette score for the output logs indicate near-perfect partitioning. Similar to the VerifyPIN case study, the high values of clustering metrics for the Nova Smart Home case study suggest that the output log and error log labels can be used reliably as features in the supervised machine learning stage.

5.2 Dataset Analysis

The results obtained by analyzing the features of the final dataset with respect to the fault type using statistical association measures explained in Section 3.4 are presented. These results determine whether or not each feature is suitable to be included in the supervised machine learning stage and the approximate predictive ability of each feature. The data analysis results also indicate the effectiveness of the framework to create a software system-specific labelled dataset.

5.2.1 VerifyPIN Dataset

The dataset generated for the VerifyPIN software, which only contains categorical features, is analyzed for feature selection. Fig. 5.1 compares the Cramer's V coefficient measured for the categorical features of the dataset using a barchart, wherein a value of 0 indicates no association and a value of 1 indicates a high association between the feature and fault type. It clearly illustrates that Label_Log and Exit_Code, with coefficient values of 0.46 and 0.39, are the best predictors of the fault type, followed by Time_Out and Label_Error, with

coefficient values of 0.28 and 0.19. The coefficient values of 0.07 and 0.02 for Card_Pin and User_Pin suggest that they are not good predictors of the fault type in this dataset. This is because the program behavior of VerifyPIN is solely dependent on the card and user PIN being equal. Once the card PIN is set, the program behavior is solely dependent on the equality of the two PINs and does not differ for different incorrect PINs entered by the user.

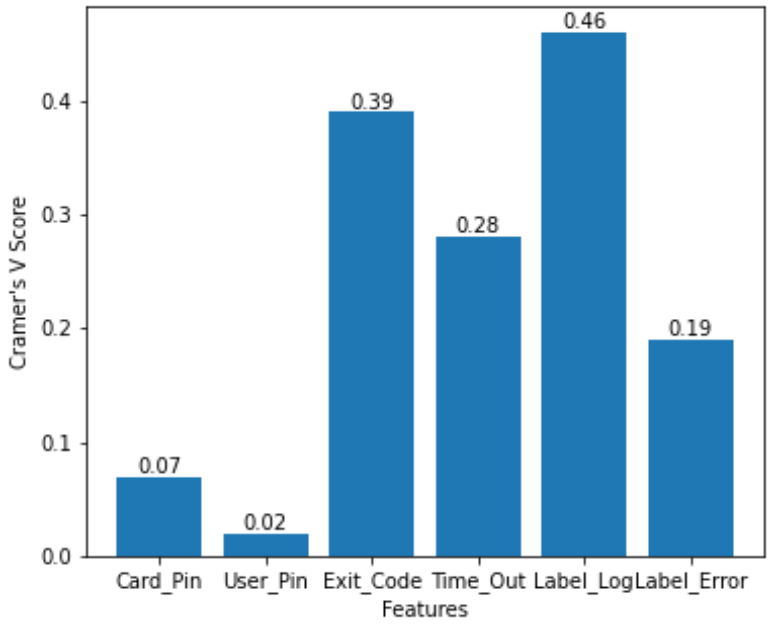

Figure 5.1: VerifyPIN: Cramer's V Scores for the Categorical Features.

Next, the mutual information scores of the features are compared. Fig. 5.2 compares the mutual information scores for each feature of the final dataset, wherein values closer to 1 indicate a higher reduction of uncertainty in predicting the fault type, whereas values closer to 0 indicate a lower reduction in the uncertainty. It reaffirms that Label_Log, Exit_Code, Time_Out, and Label_Error are good predictors of the fault type. Therefore, the application data consisting of Card_Pin and User_Pin are dropped from the dataset for training supervised machine learning models in the next stage. The application data is not important in making the fault type predictions for the VerifyPIN program. The presence of four good

predictors of the fault type in the dataset indicates the effectiveness of the labelled dataset generation stage of the framework.

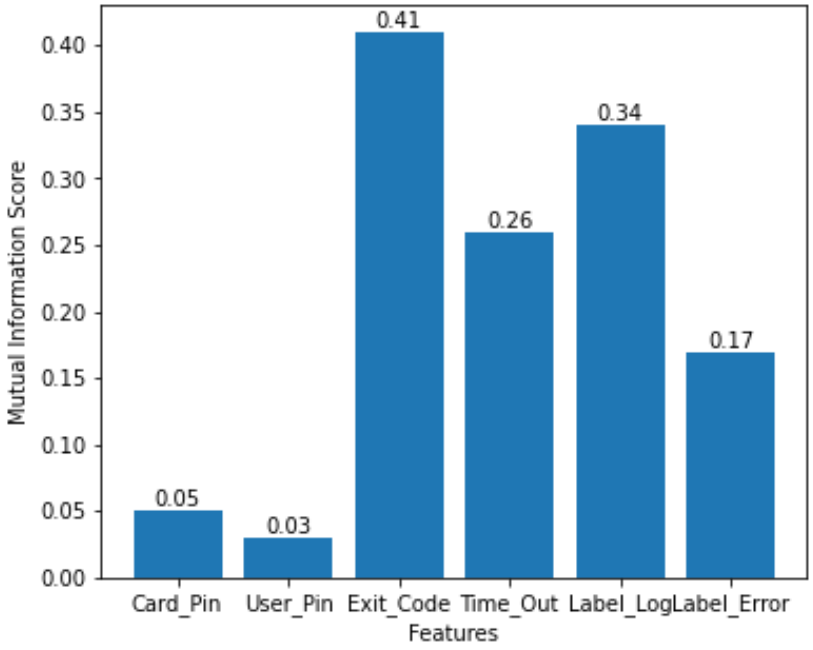

Figure 5.2: VerifyPIN: Mutual Information Scores for All Features.

5.2.2 Nova Smart Home Dataset

The categorical and numerical features of the generated Nova Smart Home dataset are analyzed for feature selection. Fig. 5.3 compares the Cramer's V coefficient measured for the categorical features of the dataset using a barchart. It clearly illustrates that all the features included in the dataset are suitable as predictors of the fault type. Label_Log is the best predictor with a score of 0.37, whereas Time_Out is the worst predictor amongst all features with a score of 0.17. Next, the Kruskal-Wallis p-values of the numerical features are compared. Fig. 5.4 compares the Kruskal-Wallis p-values for each numerical feature, wherein p-values closer to 1 indicate a higher confidence against the null hypothesis that the

feature and fault type are not related, whereas values closer to 0 indicate lower confidence against the null hypothesis.

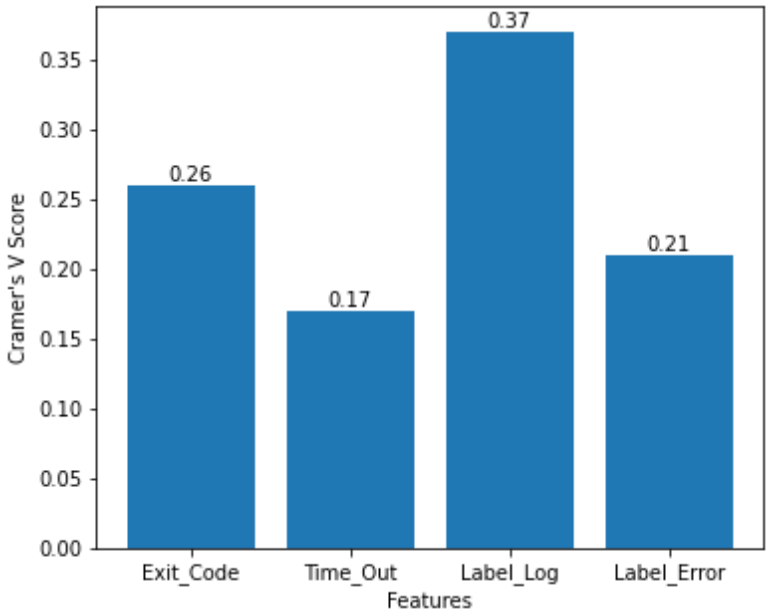

Figure 5.3: Nova Smart Home: Cramer's V Scores for the Categorical Features.

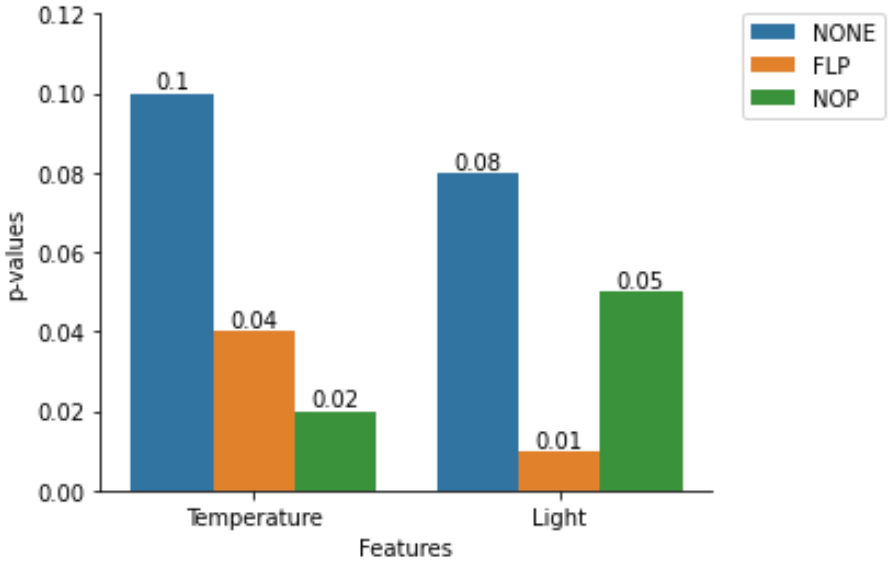

Figure 5.4: Nova Smart Home: Kruskal-Wallis p-value Scores for the Numerical Features.

It clearly illustrates that temperature and light with p-values closer to 1 have a strong statistical relationship with the fault types of NOP and FLP, whereas they have a slightly weaker statistical relationship with the NONE fault type.

This implies that both temperature and light can be considered as suitable predictors of the fault type. Fig. 5.5 compares the mutual information scores for each feature of the final dataset. Label_Log, Temperature, Light, and Exit_Code are strong predictors of the fault type with scores greater than 0.3. Label_Error and Time_Out have mutual information scores less than 0.3, but are still acceptable predictors of the fault type.

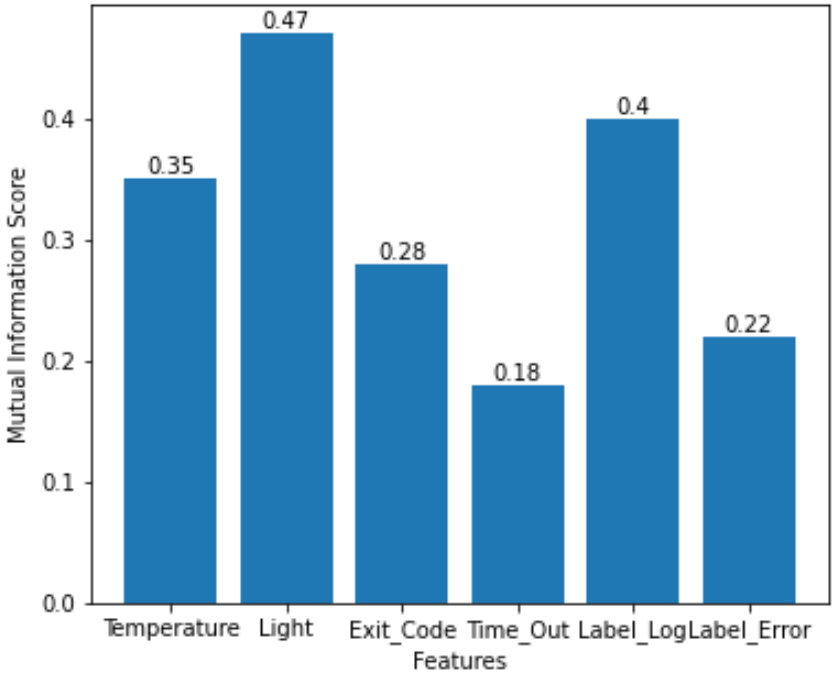

Figure 5.5: Nova Smart Home: Mutual Information Scores for All Features.

The scores of 0.35 and 0.47 for temperature and light indicate that application data is important for the fault type prediction in this dataset. This is because the multiple decision-making blocks that use application input data in the Nova Smart Home cause the software

behavior under injected faults to significantly differ for different values of temperature and light. All the features from the final dataset are included in training the supervised machine learning models in the next stage. The presence of six good predictors of the fault type in the dataset indicates the effectiveness of the labelled dataset generation stage of the framework.

5.3 Supervised Machine Learning - Training & Evaluation

This section presents the results obtained from the training and evaluation of supervised machine learning models for the detection of low-level simulated attacks on the software executables and classifying the fault type. Evaluation metrics for different types of supervised machine learning models are reported. Naive classifiers making constant predictions are used as the baseline for evaluating the usefulness of other machine learning models. Three naive classifiers are trained to make constant predictions of NOP, FLP and NONE, respectively. Different types of supervised machine learning algorithms are evaluated on the dataset. The K-Nearest Neighbors (K-NN) algorithm is chosen as a distance-based classification algorithm in predicting the fault type.

Since the K-NN algorithm also performs well with a smaller dataset, it can be used when the range of application data used to create the dataset is narrow, resulting in a small dataset. In the case of a larger dataset, K-NN algorithms require less training time than others. However, making predictions using the deployed model in the live environment requires more time compared to other supervised models. Random Forest (RF) is chosen as the first decision tree-based ensemble machine learning algorithm because it performs better and is more efficient than decision tree algorithm on larger datasets. Random Forest builds a multitude of decision trees and then uses bagging to combine the predictions from all of the models. XGBoost is the second decision tree-based ensemble machine learning

algorithm that is trained and evaluated. Similar to the Random Forest algorithm, XGBoost performs better and is more efficient than individual decision trees trained on larger datasets. XGBoost employs the boosting ensemble learning technique, as opposed to Random Forest, which employs the bagging ensemble learning technique. XGBoost builds a multitude of weak decision trees and then generates the final predictions by combining the predictions generated by each tree using the gradient boosting ensemble technique.

Finally, the Naive Bayes (NB) model is evaluated in the probabilistic classifiers category. Similar to K-NN, Naive Bayes is preferred as the probabilistic classifier because it also performs well with a smaller dataset and can handle both continuous and categorical features. Hyper-parameter optimization is not performed for the Naive Bayes classification model. The NB model is trained and evaluated using k-fold CV with k=10, which implies that ten combinations of train and test data are used to evaluate the model. K-NN, RF, XGBoost, and NB algorithms are chosen because they suit the non-linear and categorical nature of the dataset. RF and XGBoost algorithms can greatly reduce the likelihood of overfitting when combined with proper hyperparameter tuning during Nested-CV. RF and XGBoost are also preferred because they automatically select a subset of the best features while training the model.

5.3.1 VerifyPIN Dataset

The mean values of recall, precision, and f1-score for all of the machine learning models trained on the VerifyPIN dataset using Nested-CV are listed in Table. 5.3. The evaluation metrics for the Naive classifier suggest that any machine learning model with a recall, precision, and f1-score greater than 33.3%, 11.1%, and 16.7%, respectively, can be considered as a model that reduces the uncertainty in predicting the fault type. These metrics are considered as the baseline for comparing other supervised machine learning models

trained on the VerifyPIN dataset. As explained in Section 4.2.4, recall is considered as the primary metric, while precision is considered as the secondary metric. A recall value of 73% indicates that 73% of the actual injected faults were correctly classified, whereas a precision value of 85% indicates that 85% of predicted faults were actual injected faults. The maximum recall value of 73% is sufficient to achieve the primary objective of minimizing the number of incorrectly classified actual injected faults, which are the false negatives. Similarly, the maximum precision value of 85% is sufficient to achieve the secondary objective of minimizing the instances that were incorrectly classified as injected faults, which are the false positives. The correct classification of these low-level faults can enable the precise detection of low-level software effects of fault injection attacks on the VerifyPIN software system.

Table 5.3. VerifyPIN: Mean Evaluation Metrics for All Models Using 10-fold Nested Cross Validation.

Machine Learning Model	Recall	Precision	F1-score
Naive	33.3%	11.1%	16.7%
K-NN	53.4%	70.6%	44.7%
Random Forest	73.3%	85.2%	73.8%
XGB	73.4%	85.2%	73.8%
Naive Bayes	73.3%	85.2%	73.8%

Fig. 5.6 shows the evaluation results for the optimized K-NN model obtained in each fold of the Nested-CV. It illustrates that the recall, precision, and f1-score vary between 45% and 80%. The precision varies significantly for different combinations of train and test data,

whereas the recall and f1-score are relatively constant. As a result, the recall, precision, and f1-score of K-NN model have low mean values of 53.4%, 70.6%, and 44.7%, respectively. Fig. 5.7 shows the evaluation results obtained in each fold of the Nested-CV for the optimized Random Forest model. It clearly illustrates that, unlike K-NN, the evaluation metrics vary between 73.5% and 85.5% but do not vary significantly for different combinations of train and test data. As illustrated in Table. 5.3, the mean values of recall, precision and f1-score obtained for the Random Forest, XGB, and Naive Bayes models are 73.3%, 85.2% and 73.8%, respectively. The similar mean values listed in the table demonstrate that the Random Forest, XGB, and Naive Bayes models perform equally well on this dataset. The above-mentioned results demonstrate that the distance-based K-NN model outperforms the naive model but is not as good and consistent as the ensemble decision tree and probabilistic machine learning models.

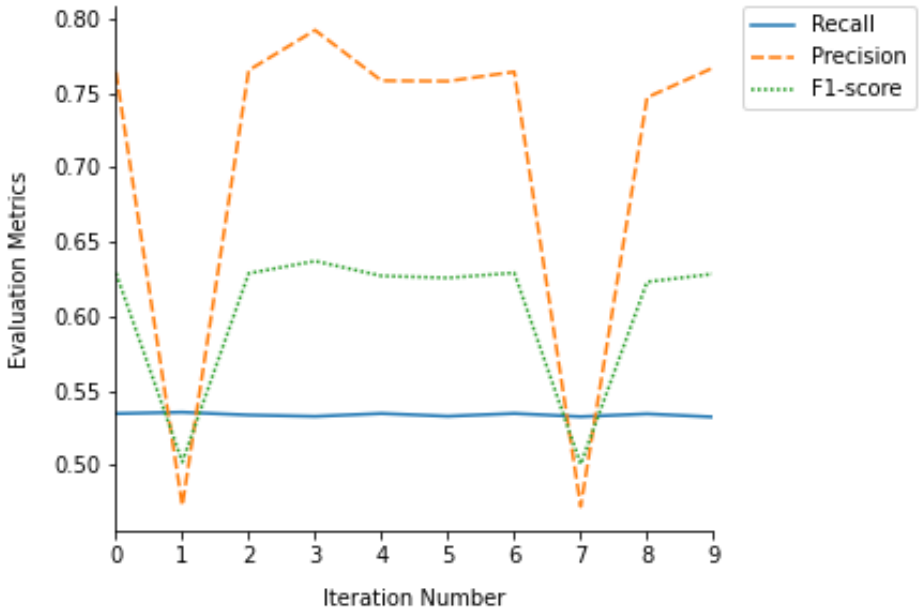

Figure 5.6: VerifyPIN: Evaluation Metrics for K-NN Using 10-fold Nested Cross Validation.

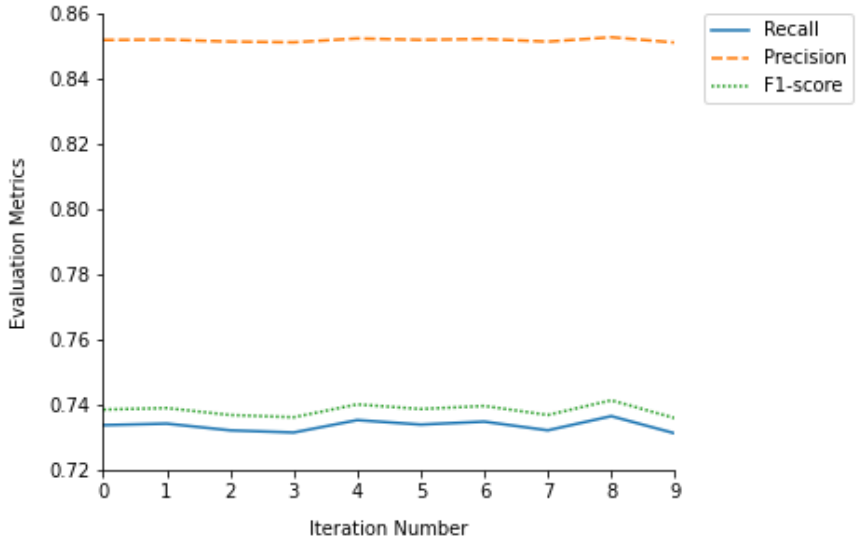

Figure 5.7: VerifyPIN: Evaluation Metrics for Random Forest Using 10-fold Nested Cross Validation.

5.3.2 Nova Smart Home Dataset

The mean values of recall, precision, and f1-score for all of the machine learning models trained on the Nova Smart Home dataset using Nested-CV are listed in Table. 5.4.

Table 5.4. Nova Smart Home: Mean Evaluation Metrics for All Models Using 10-fold Nested Cross Validation.

Machine Learning Model	Recall	Precision	F1-score
Naive	33.3%	12.9%	18.3%
K-NN	62.3%	80.9%	60.3%
Random Forest	88.2%	90.4%	85%
XGB	88.3%	90.4%	85%
Naive Bayes	84.6%	83.7%	83.5%

The evaluation metrics for the Naive classifier suggest that any machine learning model with a recall, precision, and f1-score greater than 33.3%, 12.9%, and 18.3%, respectively, can be considered as a model that reduces the uncertainty in predicting the fault type. These metrics are considered as the baseline for comparing other supervised machine learning models trained on the Nova Smart Home dataset. The high precision and recall values of supervised machine learning models in the range of 80-90% are sufficient to achieve the primary and secondary objectives of minimizing the false negatives and false positives, respectively, as explained in Section 4.2.4. For instance, a recall value of 88% indicates that 88% of the actual injected faults were correctly classified, whereas a precision value of 90% indicates that 90% of predicted faults were actual injected faults. Fig. 5.8 shows the evaluation results for the optimized Random Forest model obtained in each fold of the Nested-CV.

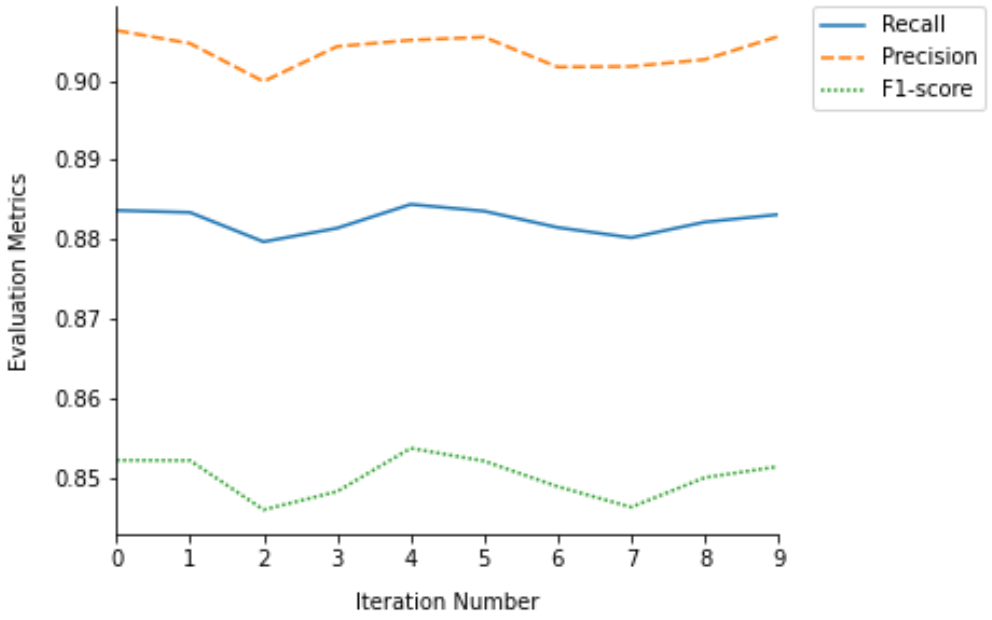

Figure 5.8: Nova Smart Home: Evaluation Metrics for Random Forest Using 10-fold Nested Cross Validation

It illustrates that the recall, precision, and f1-score vary between 0.840 and 0.910 but do not vary significantly for different combinations of train and test data. The mean values of recall, precision and f1-score obtained for the Random Forest model are 88.2%, 90.4%, and 85%, respectively.

Comparing the mean evaluation metrics of the K-NN model listed in Table. 5.4 with other models clearly demonstrates that decision tree and probability-based machine learning models outperform the distance-based K-NN model on this dataset. The recall, precision, and f1-score of the K-NN model have low mean values of 62.3%, 80.9%, and 60.3%, respectively for this dataset. The Random Forest and XGB models perform the best with mean recall, precision, and f1-score of 88.3%, 90.4%, and 85%, respectively. Random Forest and XGB, both ensemble decision tree-based classifiers, generate similar results on this dataset. The Naive Bayes classifier with mean recall, precision, and f1-score of 84.6%, 83.7%, and 83.5%, respectively, performs better than the Naive and K-NN models, but not as good as the decision tree-based classifiers. Fig. 5.9 shows the evaluation metrics for the optimized Naive Bayes model obtained in each fold of the Nested-CV. It demonstrates that the prediction performance is best during the fourth iteration, but varies slightly between different iterations, unlike the Random Forest and XGB classifiers.

Finally, the evaluation metrics for system-level threat detection implemented on the Nova Smart Home dataset are presented. Table 5.5 shows the evaluation metrics for the NOP and FLP fault models. Recall values of 85.59% and 83.85% for NOP and FLP indicate that the proposed system-level threat detection approach due to low-level simulation of fault injection attacks on software systems is highly effective. The effectiveness of the proposed approach is corroborated by high values of accuracy, precision, and f1-score. F1-scores of 86.59% and 86.67% indicate that the approach minimizes both types of incorrect predictions, false positives and false negatives.

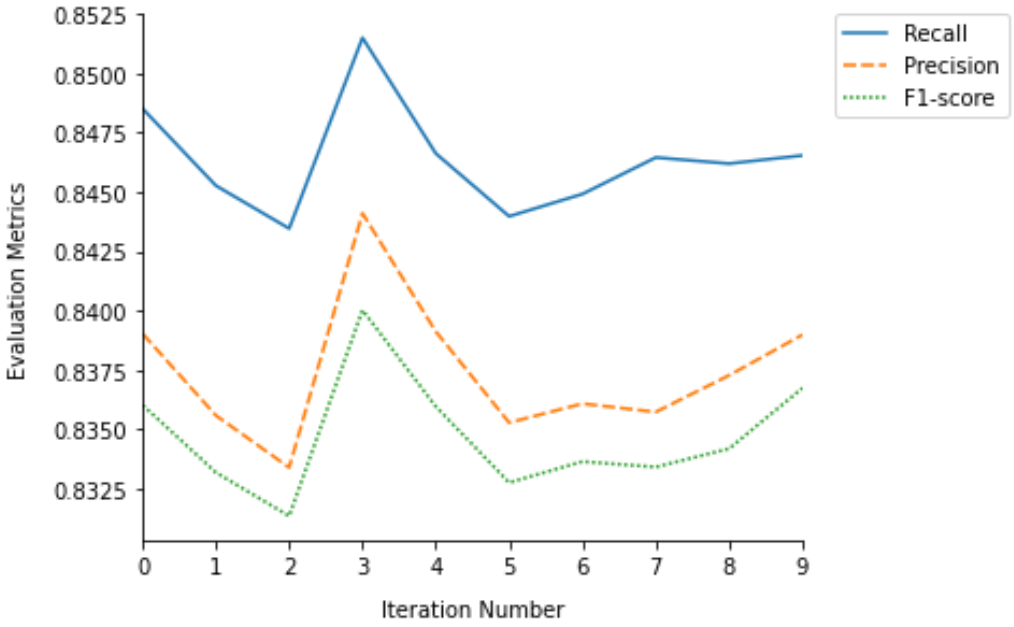

Figure 5.9: Nova Smart Home: Evaluation Metrics for Naive Bayes Using 10-fold Cross Validation

Table 5.5. Nova Smart Home: Evaluation Metrics for System-Level Threat Detection.

Fault Model	Accuracy	Precision	Recall	F1 Score
NOP	85.59%	87.81%	85.59%	86.59%
FLP	83.85%	90.53%	83.85%	86.67%

5.4 Discussion

This section analyzes the machine learning evaluation results presented in Section 5.3. The mean evaluation metrics for the two datasets listed in Table. 5.3 and Table. 5.4 demonstrate that all the trained models perform significantly better than the naive model across all evaluation parameters. Comparing the mean evaluation metrics of the K-NN model to other classifiers demonstrates that the K-NN model cannot be relied upon completely due to the

low values obtained as a result of large deviations in precision for different combinations of train and test data. The large deviation in precision is also evident in Fig. 5.6. Unlike K-NN, the deviation in the evaluation metrics for Random Forest, XGB, and Naive Bayes models is minimal, making them more reliable than the K-NN model. Further, the average prediction time for K-NN models is 1.86s for 10,000 samples, whereas Random Forest, XGB, and Naive Bayes require around 0.26s. Thus, Random Forest, XGB and Naive Bayes outperform K-NN and can produce faster predictions in the live environment. The probability-based Naive Bayes model outperforms the distance-based K-NN model, but is not as good as decision-tree based Random Forest and XGBoost models.

Finally, all the supervised machine learning models are trained and evaluated for different percentages of train and test data using the optimized hyperparameters. Fig. 5.10 and Fig. 5.11 show the evaluation metrics for the Random Forest classifiers trained using various ratios of train and test data on the VerifyPIN and Nova Smart Home datasets, respectively.

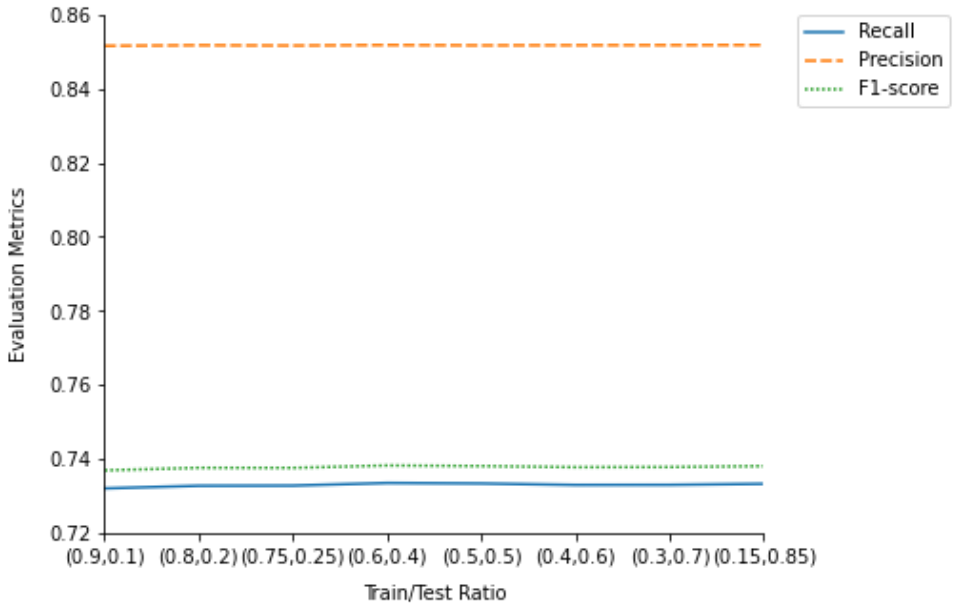

Figure 5.10: VerifyPIN: Evaluation Metrics for Random Forest Using Various Train and Test Data Ratios

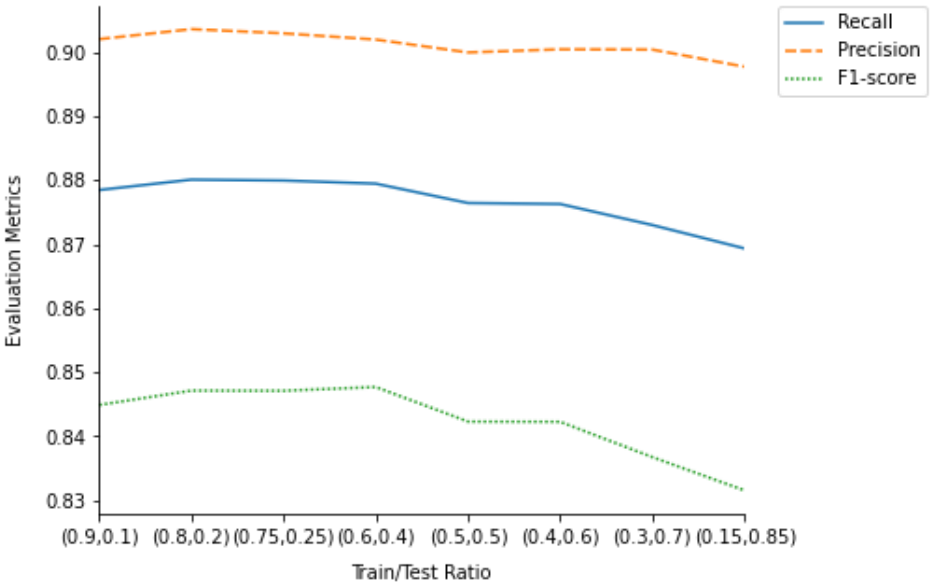

Figure 5.11: Nova Smart Home: Evaluation Metrics for Random Forest Using Various Train and Test Data Ratios

The common observation from the graphs is that as the percentage of train data decreases, the recall, precision, and f1-score vary by a small margin. This further strengthens the statistical significance of the results. Table. 5.5 and Table. 5.6 list the evaluation metrics for all the classifiers trained on the two datasets using various ratios of train and test data. Overall, the results listed in Tables 5.3-5.6 demonstrate that the Random Forest and XGBoost classifiers perform the best across the two datasets. The precision and recall values of these classifiers are sufficient to minimize the number of incorrectly predicted low-level faults and correctly classify the majority of actual low-level faults.

Further evaluation is carried out to measure the importance of software features and application data in predicting low-level fault injections in the software. The feature importance for the random forest model is computed to demonstrate the overall usefulness of each feature in generating predictions for both case studies. Fig. 5.12 and Fig. 5.13 show the feature importance of each feature in making the fault type predictions on the test data.

Table 5.6. VerifyPIN: Mean Evaluation Metrics for All Models Using Various Train and Test Data Ratios.

Machine Learning Model	Recall	Precision	F1-score
K-NN	73.1%	84.9%	73.6%
Random Forest	73.3%	85.2%	73.8%
XGB	73.3%	85.2%	73.8%
Naive Bayes	73.3%	85.2%	73.8%

Table 5.7. Nova Smart Home: Mean Evaluation Metrics for All Models Using Various Train and Test Data Ratios.

Machine Learning Model	Recall	Precision	F1-score
K-NN	87.1%	88.6%	84%
Random Forest	87.7%	90.1%	84.2%
XGB	88.1%	90.3%	84.9%
Naive Bayes	84.7%	83.8%	83.6%

For the VerifyPIN case study, it illustrates that all features in the dataset contribute well to fault type prediction; however, Label_Log, Label_Error and Exit_Code are the most important features in predicting the fault type. For the Nova Smart Home case study, it illustrates that Label_Log is the most important feature in predicting the fault type. Temperature and Light are also important features when predicting the fault type, whereas Time_Out and Exit_Code are the least important features. This provides preliminary support for the research hypothesis that run-time software properties and application data can be combined to enhance prediction performance for certain types of software systems.

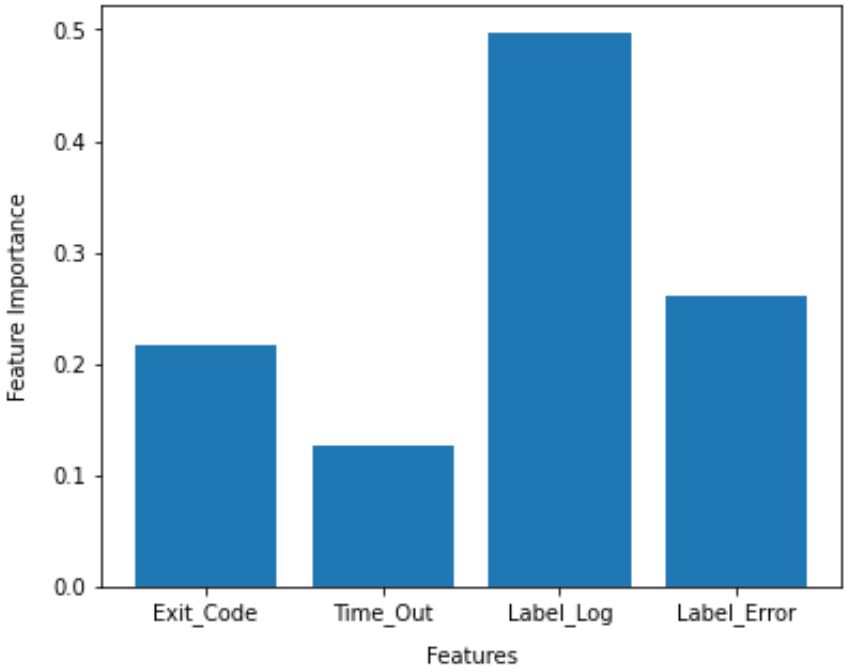

Figure 5.12: VerifyPIN: Feature Importance for Random Forest.

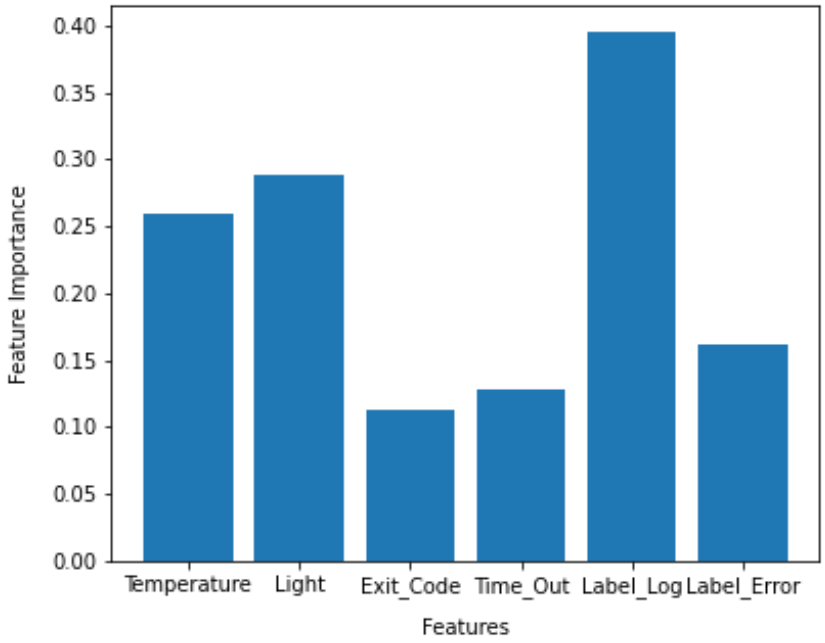

Figure 5.13: Nova Smart Home: Feature Importance for Random Forest.

To provide additional support for the hypothesis, three separate random forest models are trained and evaluated on the Nova Smart Home dataset. Model 1 is trained using the

application data as the features and the fault type as the label. Model 2 is trained using the software features and fault type as the label. Model 3 is trained by combining application data and software properties as the features. A heatmap is generated for each model in order to analyze the results in greater detail and understand the classification performance for each class in this multiclass classification problem. Fig. 514, Fig. 5.15, and Fig. 5.16 represent the classification performance of Models 1, 2, and 3 for the classes FLP, NONE, and NOP, respectively. It is evident that the prediction performance of Model 3 is superior than Models 1 and 2, primarily due to the improvement in the classification of NOP instances. Models 1 and 2 yield precision values of 15% and 88%, and extremely low recall values of 2.6% and 3% for the NOP class. On the other hand, Model 3 yields precision and recall values of 97% and 20% for the NOP class. The low value of recall can be attributed to a high number of false negatives for the NOP class, which is an existing limitation. This could be due to the relatively small number of NOP instances in the dataset. Though the recall for the NOP class remains low, it is a significant improvement over the separate use of software features and application data. This proves the rationale presented in Section 3.2 for the inclusion of application data and software features in the dataset.

Fig. 5.17 compares the overall recall, precision, and f1-score for Model 1, Model 2, and Model 3. It is evident that Model 3 performs significantly better than Models 1 and 2. For example, Model 3 provides an overall recall of 0.882, whereas Models 1 and 2 provide recall values of 0.386 and 0.653, respectively. This conclusively proves that combining application data and software properties leads to an improved prediction and classification of the fault type in certain types of software systems. Application data should be included as features for software systems whose behavior varies greatly depending on the application input value. Sensor measurements such as temperature, light intensity, obstacle detection readings, and other continuous variables are examples of this type of application input.

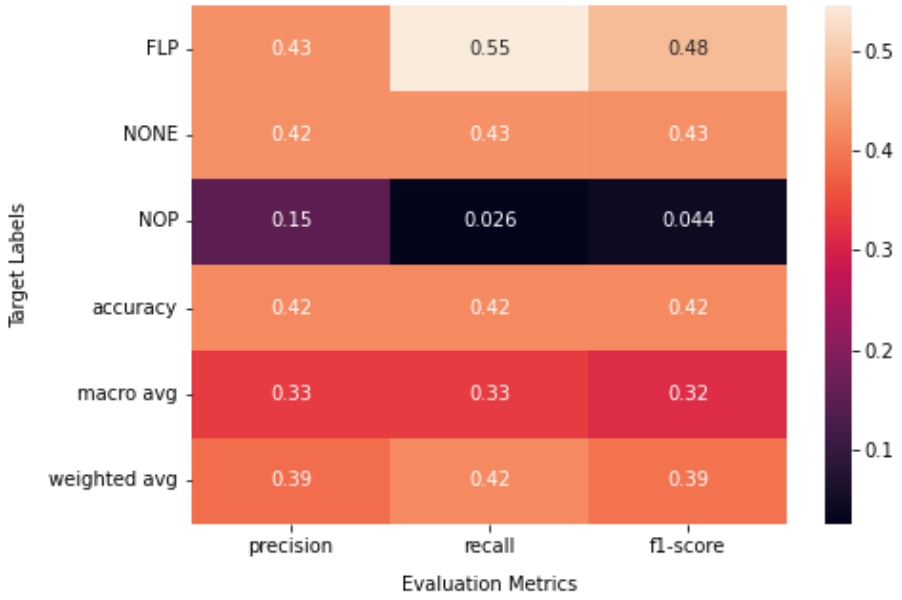

Figure 5.14: Nova Smart Home: Heatmap for Model Trained Using Application Data.

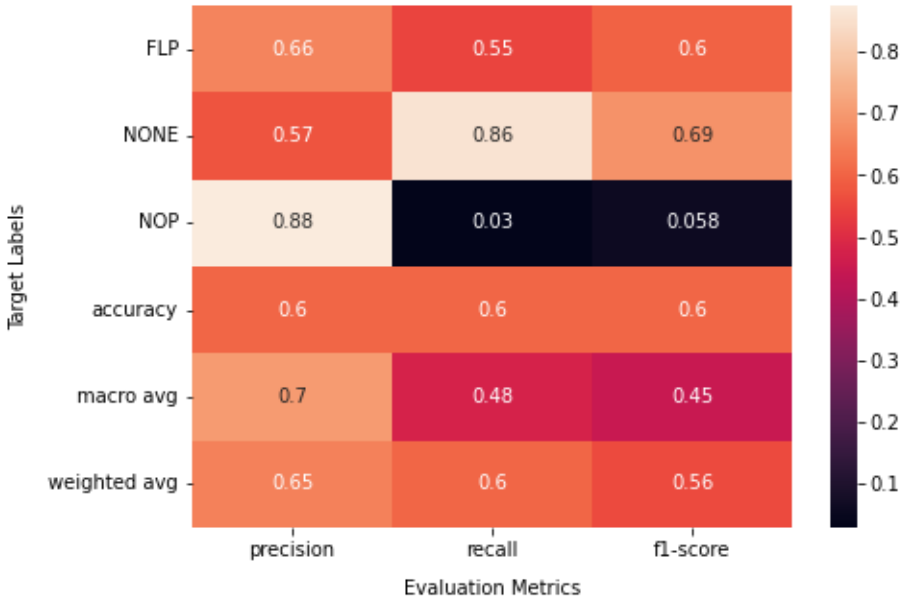

Figure 5.15: Nova Smart Home: Heatmap for Model Trained Using Software Features.

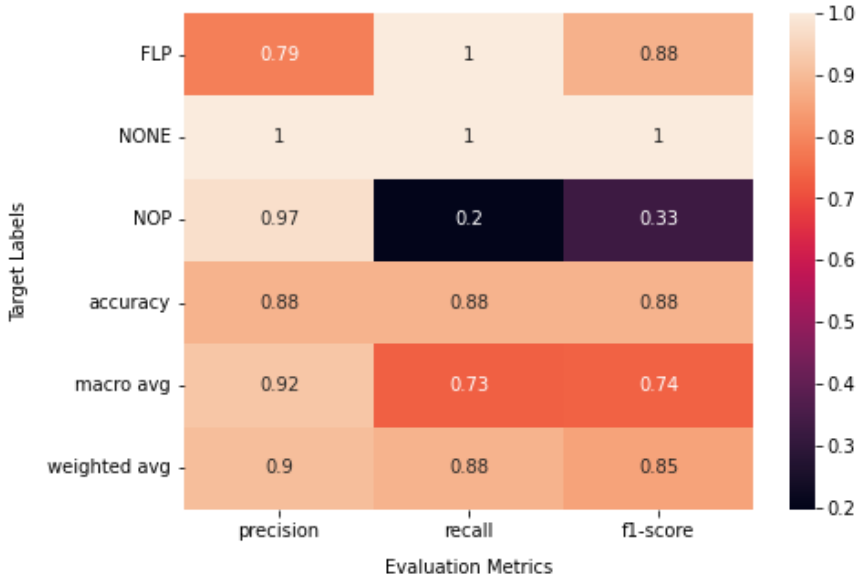

Figure 5.16: Nova Smart Home: Heatmap for Model Trained by Combining Application Data and Software Features.

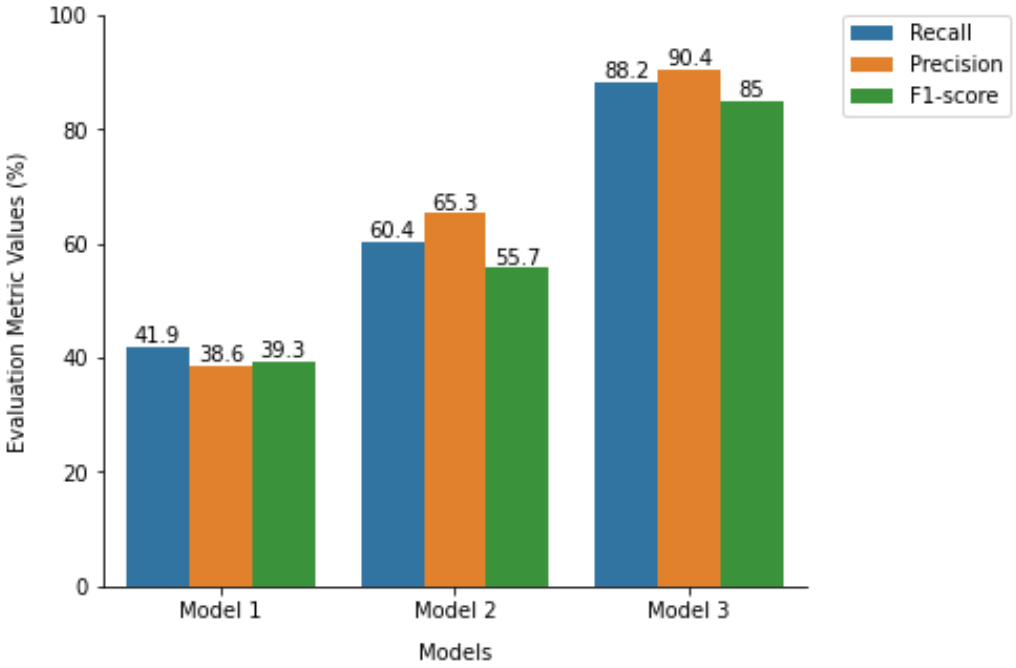

Figure 5.17: Evaluating Usefulness of Application data and Software properties.

The analysis of the results demonstrates the effectiveness of the proposed dataset generation framework and supervised machine learning-based framework in the detection of low-level fault injections in software systems. The modified chaos duck software fault injection tool can be utilized to generate a labelled dataset for any software system with an executable, making the proposed framework generic. Once the dataset is generated, the proposed supervised machine learning steps can be followed to generate a model capable of detecting and classifying the low-level fault injections.

Finally, the proposed supervised machine learning-based framework is compared qualitatively and quantitatively to the relevant literature mentioned under related work. In comparison to the proposed framework which detects low-level fault injections in software systems, the method proposed by Wei et al. [9] formulates the detection of fault injection attacks as a system design problem and utilizes a fault detection scheme based on an unsupervised evolutionary algorithm. The method proposed in [9] is only applicable to AES encryption software, whereas the proposed supervised machine learning-based framework and labelled dataset generation in this thesis is applicable to majority of the software systems. The method proposed by Riviere et al. [37] utilizes the EFS to simulate control flow fault injections on the VerifyPIN software used in smart cards. They achieve a detection rate of 69.2%, whereas the proposed framework in this thesis achieves a detection rate of 74%, which is an improvement by 7%. Moreover, the proposed framework utilizes multiple fault models simulating control flow, instruction skip, and other software effects of fault injection attacks. Koylu et al. [27] present a RNN based program flow alteration detection due to fault injection attacks on the RSA encryption software. In comparison to the average accuracy of 83.7%, the proposed framework in this thesis achieves an average accuracy of 88.2%. In comparison to all the methods mentioned above, the proposed framework can be utilized to

generate supervised machine learning models for the detection and classification of low-level fault injections in the live environment of any software system.

5.5 Threats to Validity

This section describes the various threats to the implementation of the proposed framework and the validity of the results obtained from the two case studies. The threats to validity are categorized into internal and external threats. The following internal validity threats identify the factors affecting the reliability of the current framework implementation and results obtained from the case studies:

- The usefulness of the generated dataset by the framework for low-level fault injection detection is highly dependent on the application input data utilized in the labelled dataset generation stage. For instance, the range of temperature readings in the Nova Smart Home software was determined between -30°C and -100°C using knowledge about the software. This implies that the selection of application input data used in the labelled dataset generation stage requires the assistance of a domain expert for complicated and large-scale software systems. Alternatively, multiple iterations of dataset and model generation may be performed using different range of application input data for each iteration.
- The low-level fault injection detection results are highly dependent on any unsupervised models and techniques used in the data preprocessing stage. For instance, in the case studies, accuracy of the labels generated for the output and error logs played an important role in the detection performance. The accuracy of the log labels was ensured using clustering metrics such as homogeneity score. For software systems generating huge logs, the use of advanced unsupervised clustering

techniques and log filtering may be required to generate reliable log labels for the supervised machine learning stage.

- The ability of the machine learning model to differentiate between various low-level fault types is also dependent on the fault injection points utilized to generate the labelled dataset. For example, the low recall values presented in Section 5.3.2 for the NOP fault type can be attributed to this factor due to the small number of NOP instances in the dataset. As explained in Section 3.1, two and three iterations of dataset and model generation were performed for the VerifyPIN and Nova Smart Home case studies to ensure the usefulness and reliability of the selected machine learning model for different low-level fault types.
- Nested-CV was utilized to ensure the statistical significance and reliability of the results obtained from the case studies.

The following external validity threats identify the factors affecting the scalability and adaptability of the framework implementation, and the generalizability and reproducibility of the results obtained from the case studies:

- The case studies repeated using the same injection points and application input data should produce the same results as the first time. However, changing these parameters using the fault injection control block may generate different results.
- The labelled dataset and corresponding supervised machine learning model generated for a specific software system may not produce similar results for other similar software systems, or even if the same software system is deployed in a completely new environment with different operating conditions. This may necessitate regeneration of the dataset and model, and poses a significant threat to the scalability and adaptability of the framework.

- The machine learning model generated using the simulation-based labelled dataset has not been evaluated in the live environment of a software system. Evaluating the model in a real-world software system will aid in thoroughly generalizing and comparing the results to other frameworks in literature.

5.6 Summary

This chapter has presented the evaluation results for the dataset generation stage and supervised machine learning stage of the framework for detecting and classifying low-level fault injections in software systems at an instruction-level. The datasets generated in the case studies were analyzed utilizing data analysis techniques to select the appropriate features and obtain the final dataset. The supervised machine learning models trained using the final dataset were evaluated and compared to demonstrate their effectiveness and combination of software properties and application data for certain software systems. Additionally, the proposed framework and results were compared to the relevant literature.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis presents a supervised machine learning-based framework for detecting low-level fault injections in any software system. The framework utilizes an instruction-level software fault injection tool to implement the software system-specific labelled dataset generation stage and enable the use of supervised machine learning. The labelled dataset can subsequently be used to train a supervised machine learning model for detecting low-level fault injections in software systems and classifying the type of fault introduced into the software. The model generated by the framework can be utilized for the detection of low-level effects of fault injection attacks in the live environment of software systems. The use of an instruction-level software fault injection tool enables the low-level detection of various attack effects such as control flow modifications due to processor instruction skip and abnormal application software behavior due to manipulation of the input data.

A case study on a safety-critical pin verification software is conducted to demonstrate how run-time software properties can be used to detect and classify low-level fault injections in software systems. The low-level fault injection predictions made by the proposed framework are more accurate and precise due to the use of supervised machine learning, and the detection mechanism can be made more resistant to adversarial machine learning and false data injection by training the machine learning model with a labelled dataset containing both true and false data instances with the appropriate target labels. Another case study on a home automation IoT software demonstrates that application data can be combined with run-

time software properties to detect and classify low-level fault injections with greater precision for software whose behavior varies significantly depending on the application input value. The use of the generated datasets to train a model capable of detecting system-level threats as a result of low-level fault injections is also presented.

6.2 Future Work

This section presents the future possibilities to extend and improve this research.

- Additional software properties can be included in the dataset to improve prediction performance and differentiate the abnormal software behavior due to fault injection attacks from other types of software anomalies such as internal software malfunctions. These are the software properties that are affected by fault injection attacks, but not software anomalies.
- The inclusion of application data for certain types of software systems can also differentiate between low-level software effects of fault injection attacks and other software malfunctions. Software behavior in the presence of low-level fault injections is more dependent on application input data values than during software malfunction.
- The possibility of detecting attack effects from sources such as the operating system and SQL injection by including appropriate high-level software properties should be investigated. Further evaluation using additional software properties and application input data should be performed to evaluate the above.
- Instruction-level features such as assembly instruction execution count, register status, and cache status give a better indication of low-level fault injections, but cannot be easily monitored at run-time. Future work should focus on methods to

easily monitor such low-level features or examine the use of additional high-level generic software features that can improve the predictions.

- The machine learning model generated by the proposed framework should be further evaluated in the live environment of a software system, either replicating actual fault injection attacks or using emulation-based methods proposed in the literature. This would enable the comparison of the model generated using the proposed simulation-based approach and the model trained using data from the live environment of a software system.
- Evaluation of the performance and feasibility of the proposed framework on a software system generating multiple raw features that cannot be used directly in the machine learning model should be performed. For instance, evaluation in the live environment of a software system that generates logs of varying sizes and formats.
- Further experiments need to be performed to evaluate the resistance of the model generated by the proposed framework to adversarial machine learning and false data injection by combining supervised machine learning with techniques such as adversarial training with perturbation or noise, gradient masking, and input regularization.

Bibliography

[1] B. Yuce, P. Schaumont, and M. F. Witteman, “Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation,” *Journal of Hardware and Systems Security*, vol. 2, pp. 111–130, 2018, doi: https://doi.org/10.1007/s41635-018-0038-1.

[2] A. Jha, “Introduction to Fault Injection Attack (FI)).” Dec. 2020. Accessed: Dec. 15, 2021. [Online]. Available: https://payatu.com/blog/asmita-jha/fault-injection-basics

[3] S. Delarea and Y. Oren, “Practical, Low-Cost Fault Injection Attacks on Personal Smart Devices,” *Applied Sciences (Switzerland)*, vol. 12, no. 1, 2022, doi: 10.3390/app12010417.

[4] N. Khoshavi, C. Broyles, Y. Bi, and A. Roohi, “Fiji-FIN: A Fault Injection Framework on Quantized Neural Network Inference Accelerator,” in *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2020, pp. 1139–1144. doi: 10.1109/ICMLA51294.2020.00183.

[5] B. M. Padmanabhuni and H. B. K. Tan, “Buffer Overflow Vulnerability Prediction from x86 Executables Using Static Analysis and Machine Learning,” in *2015 IEEE 39th Annual Computer Software and Applications Conference*, 2015, vol. 2, pp. 450–459. doi: 10.1109/COMPSAC.2015.78.

[6] A. Gangolli, Q. H. Mahmoud, and A. Azim, “A Systematic Review of Fault Injection Attacks on IoT Systems,” *Electronics*, vol. 11, no. 13, 2022, doi: 10.3390/electronics11132023.

[7] F. Pasqualetti, F. Dörfler, and F. Bullo, “Attack Detection and Identification in Cyber-Physical Systems,” *IEEE Transactions on Automatic Control*, vol. 58, no. 11, pp. 2715–2729, 2013, doi: 10.1109/TAC.2013.2266831.

[8] T. Ç. Köylü, C. R. W. Reinbrecht, S. Hamdioui, and M. Taouil, “RNN-Based detection of fault attacks on RSA,” in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, pp. 1–5. doi: 10.1109/iscas45731.2020.9180708.

[9] W. Jiang, L. Wen, J. Zhan, and K. Jiang, “Design optimization of confidentiality-critical cyber physical systems with fault detection,” *Journal of Systems Architecture*, vol. 107, p. 101739, 2020, doi: https://doi.org/10.1016/j.sysarc.2020.101739.

[10] J. Li, Y. Yang, J. S. Sun, K. Tomsovic, and H. Qi, “ConAML: Constrained Adversarial Machine Learning for Cyber-Physical Systems,” in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 52–66. doi: 10.1145/3433210.3437513.

[11] T. Given-Wilson, N. Jafri, and A. Legay, “Combined software and hardware fault injection vulnerability detection,” *Innovations in Systems and Software Engineering*, vol. 16, no. 2, pp. 101–120, 2020.

[12] I. Zavalyshyn, T. Given–Wilson, A. Legay, R. Sadre, and E. Rivière, “Chaos Duck: A Tool for Automatic IoT Software Fault-Tolerance Analysis,” in *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, 2021, pp. 46–55. doi: 10.1109/SRDS53918.2021.00014.

[13] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. de Choudens, “FISSC: A fault injection and simulation secure collection,” in *International Conference on Computer Safety, Reliability, and Security*, 2016, pp. 3–11.

[14] Sinova, “Nova Smart Home: A Smart Home based on Raspberry Pi and Arduino boards.” Accessed: Feb. 22, 2022. [Online]. Available: https://github.com/SIN0VA/Nova-Smart-Home

[15] B. Hettwer, S. Gehrer, and T. Güneysu, “Applications of machine learning techniques in side-channel attacks: a survey,” *Journal of Cryptographic Engineering*, vol. 10, no. 2, pp. 135–162, 2020.

[16] T. Given-Wilson, N. Jafri, and A. Legay, “The state of fault injection vulnerability detection,” in *International Conference on Verification and Evaluation of Computer and Communication Systems*, 2018, pp. 3–21.
[17] A. Qasem, P. Shirani, M. Debbabi, L. Wang, B. Lebel, and B. L. Agba, “Automatic Vulnerability Detection in Embedded Devices and Firmware: Survey and Layered Taxonomies,” *ACM Comput. Surv.*, vol. 54, no. 2, Mar. 2021, doi: 10.1145/3432893.
[18] Z. Kazemi, M. Fazeli, D. Hely, and V. Beroulle, “Hardware Security Vulnerability Assessment to Identify the Potential Risks in A Critical Embedded Application,” in *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2020, pp. 1–6. doi: 10.1109/IOLTS50870.2020.9159739.
[19] S. Di Carlo *et al.*, “Cross-layer early reliability evaluation: Challenges and promises,” in *Proceedings of the 2014 IEEE 20th International On-Line Testing Symposium, IOLTS 2014*, 2014, pp. 228–233. doi: 10.1109/IOLTS.2014.6873704.
[20] L. Dureuil, M.-L. Potet, P. de Choudens, C. Dumas, and J. Clédière, “From Code Review to Fault Injection Attacks: Filling the Gap Using Fault Model Inference,” in *Smart Card Research and Advanced Applications*, 2016, pp. 107–124.
[21] D. Cotroneo, L. De Simone, P. Liguori, and R. Natella, “ProFIPy: Programmable Software Fault Injection as-a-Service,” in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020, pp. 364–372. doi: 10.1109/DSN48063.2020.00052.
[22] T. Given-Wilson, N. Jafri, J.-L. Lanet, and A. Legay, “An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries and Case Study on PRESENT,” in *2017 IEEE Trustcom/BigDataSE/ICESS*, 2017, pp. 293–300. doi: 10.1109/Trustcom/BigDataSE/ICESS.2017.250.
[23] H. Liao and C. Gebotys, “Methodology for EM Fault Injection: Charge-based Fault Model,” in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 256–259. doi: 10.23919/DATE.2019.8715150.
[24] S. Jha *et al.*, “ML-Based Fault Injection for Autonomous Vehicles: A Case for Bayesian Fault Injection,” in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019, pp. 112–124. doi: 10.1109/DSN.2019.00025.
[25] D. Cotroneo, L. De Simone, P. Liguori, and R. Natella, “ProFIPy: Programmable Software Fault Injection as-a-Service,” in *Proceedings - 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020*, 2020, pp. 364–372. doi: 10.1109/DSN48063.2020.00052.
[26] L. Rivière, M.-L. Potet, T.-H. Le, J. Bringer, H. Chabanne, and M. Puys, “Combining High-Level and Low-Level Approaches to Evaluate Software Implementations Robustness Against Multiple Fault Injection Attacks,” in *Foundations and Practice of Security*, 2015, pp. 92–111.
[27] T. Ç. Köylü, C. R. W. Reinbrecht, S. Hamdioui, and M. Taouil, “RNN-Based detection of fault attacks on RSA,” in *Proceedings - IEEE International Symposium on Circuits and Systems*, 2020, vol. 2020-Octob, pp. 1–5. doi: 10.1109/iscas45731.2020.9180708.
[28] H. Khosrowjerdi, K. Meinke, and A. Rasmusson, “Virtualized-Fault Injection Testing: A Machine Learning Approach,” in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 297–308. doi: 10.1109/ICST.2018.00037.
[29] M. Karpovsky, K. J. Kulikowski, and A. Taubin, “Robust protection against fault-injection attacks on smart cards implementing the advanced encryption standard,” in *Proceedings of the International Conference on Dependable Systems and Networks*, 2004, pp. 93–101. doi: 10.1109/dsn.2004.1311880.

[30] A. Höller, A. Krieg, T. Rauter, J. Iber, and C. Kreiner, “QEMU-Based Fault Injection for a System-Level Analysis of Software Countermeasures Against Fault Attacks,” in *2015 Euromicro Conference on Digital System Design*, 2015, pp. 530–533. doi: 10.1109/DSD.2015.79.
[31] G. Lacombe, D. Feliot, E. Boespflug, and M.-L. Potet, “Combining Static Analysis and Dynamic Symbolic Execution in a Toolchain to detect Fault Injection Vulnerabilities,” Sep. 2021.
[32] N. Khoshavi, C. Broyles, Y. Bi, and A. Roohi, “Fiji-FIN: A Fault Injection Framework on Quantized Neural Network Inference Accelerator,” in *Proceedings - 19th IEEE International Conference on Machine Learning and Applications, ICMLA 2020*, 2020, pp. 1139–1144. doi: 10.1109/ICMLA51294.2020.00183.
[33] Y. Zhang, B. Liu, and Q. Zhou, “A dynamic software binary fault injection system for real-time embedded software,” in *ICRMS’2011 - Safety First, Reliability Primary: Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety*, 2011, pp. 676–680. doi: 10.1109/ICRMS.2011.5979375.
[34] M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil, “Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections,” in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, 2014, pp. 213–222. doi: 10.1109/ICST.2014.34.
[35] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization, CGO*, 2004, pp. 75–86. doi: 10.1109/CGO.2004.1281665.
[36] L. Riviére, M. L. Potet, T. H. Le, J. Bringer, H. Chabanne, and M. Puys, “Combining high-level and low-level approaches to evaluate software implementations robustness against multiple fault injection attacks,” in *Foundations and Practice of Security*, 2015, pp. 92–111. doi: 10.1007/978-3-319-17040-4_7.
[37] L. Rivière, J. Bringer, T.-H. Le, and H. Chabanne, “A novel simulation approach for fault injection resistance evaluation on smart cards,” in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2015, pp. 1–8. doi: 10.1109/ICSTW.2015.7107460.
[38] A. Papadimitriou, K. Nomikos, M. Psarakis, E. Aerabi, and D. Hely, “You can detect but you cannot hide: Fault Assisted Side Channel Analysis on Protected Software-based Block Ciphers,” in *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2020, pp. 1–6. doi: 10.1109/DFT50435.2020.9250870.
[39] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, “Automatic Inference of Search Patterns for Taint-Style Vulnerabilities,” in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 797–812. doi: 10.1109/SP.2015.54.
[40] J. Grycel and P. Schaumont, “SimpliFI: Hardware Simulation of Embedded Software Fault Attacks,” *Cryptography*, vol. 5, no. 2, p. 15, Jun. 2021, doi: 10.3390/cryptography5020015.
[41] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, “SymPLFIED: Symbolic program-level fault injection and error detection framework,” in *Proceedings of the International Conference on Dependable Systems and Networks*, 2008, pp. 472–481. doi: 10.1109/DSN.2008.4630118.
[42] A. Chhabra, A. Roy, and P. Mohapatra, “Strong Black-box Adversarial Attacks on Unsupervised Machine Learning Models.” arXiv, 2019. doi: 10.48550/ARXIV.1901.09493.
[43] J. Li, Y. Yang, J. S. Sun, K. Tomsovic, and H. Qi, “ConAML: Constrained Adversarial Machine Learning for Cyber-Physical Systems,” in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021, pp. 52–66. doi:

10.1145/3433210.3437513.
[44] F. Tramer and D. Boneh, “Adversarial Training and Robustness for Multiple Perturbations,” in *Advances in Neural Information Processing Systems*, 2019, vol. 32. Accessed: Jan. 18, 2022. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/5d4ae76f053f8f2516ad12961ef7fe97-Paper.pdf
[45] H. Zhang and J. Wang, “Defense against adversarial attacks using feature scattering-based adversarial training,” *Advances in Neural Information Processing Systems*, vol. 32, 2019.
[46] T. Miyato, S. Maeda, M. Koyama, and S. Ishii, “Virtual adversarial training: a regularization method for supervised and semi-supervised learning,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 41, no. 8, pp. 1979–1993, 2018.
[47] J. Richter-Brockmann, A. R. Shahmirzadi, P. Sasdrich, A. Moradi, and T. Güneysu, “Fiver--robust verification of countermeasures against fault injections,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 447–473, 2021.
[48] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson, “Formal verification of a software countermeasure against instruction skip attacks,” *Journal of Cryptographic Engineering*, vol. 4, no. 3, pp. 145–156, 2014, doi: https://doi.org/10.1007/s13389-014-0077-7.
[49] J.-B. Bréjon, K. Heydemann, E. Encrenaz, Q. Meunier, and S.-T. Vu, “Fault Attack Vulnerability Assessment of Binary Code,” in *Proceedings of the Sixth Workshop on Cryptography and Security in Computing Systems*, 2019, pp. 13–18. doi: 10.1145/3304080.3304083.
[50] S. Nashimoto, N. Homma, Y. Hayashi, J. Takahashi, H. Fuji, and T. Aoki, “Buffer overflow attack with multiple fault injection and a proven countermeasure,” *Journal of Cryptographic Engineering*, vol. 7, no. 1, pp. 35–46, 2017, doi: https://doi.org/10.1007/s13389-016-0136-3.
[51] N. A. Quynh, “Capstone : Next-Gen Disassembly Framework,” *Black Hat USA*, vol. 5, no. 2, pp. 1–53, 2014.
[52] Chenoya, “Software Implemented Fault Injection Tool.” Accessed: Feb. 22, 2022. [Online]. Available: https://github.com/chenoya/swifi-tool
[53] R. Rehurek and P. Sojka, “Software Framework for Topic Modelling with Large Corpora,” in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, May 2010, pp. 45–50.
[54] P. Virtanen *et al.*, “{SciPy} 1.0: Fundamental Algorithms for Scientific Computing in Python,” *Nature Methods*, vol. 17, pp. 261–272, 2020, doi: 10.1038/s41592-019-0686-2.
[55] F. Pedregosa *et al.*, “Scikit-learn: Machine Learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
[56] M. L. Waskom, “seaborn: statistical data visualization,” *Journal of Open Source Software*, vol. 6, no. 60, p. 3021, 2021, doi: 10.21105/joss.03021.
[57] F. da Rosa, L. Ost, and R. Reis, “Extensive Soft Error Evaluation,” in *Soft Error Reliability Using Virtual Platforms: Early Evaluation of Multicore Systems*, Cham: Springer International Publishing, 2020, pp. 71–88. doi: 10.1007/978-3-030-55704-1_5.

Appendix

Appendix A. Source Code

The source code of the original chaos duck instruction-level software fault injection tool can be found at the below link:

https://github.com/zavalyshyn/chaosduck

Source code file of the python script file for the modified chaos duck tool is provided below. The tool begins by extracting the assembly instructions from the software executable and injects faults into the extracted instructions using the fault models. The command line arguments for launching the python script include the executable file name, the instruction set architecture (arm or x86), and the software fault models to be utilized for injecting the faults (NOP, FLP, JMP, and ZERO). The chaos duck tool internally uses the swifi tool to inject faults, which has not been included here. The complete code for the modified chaos duck software fault injection tool, the executables used for the case studies along with the generated datasets, and the Jupyter notebooks for machine learning can be found at:

https://github.com/aakashgangolli3112/software-fault-injection

A1. chaosduck.py

```
import sys, os, shlex, time, csv, shutil, traceback, random, argparse,
multiprocessing.pool
from elftools.elf.elffile import ELFFile
from elftools.common.exceptions import ELFError
from capstone import *
from capstone.x86 import *
from pathlib import Path
from subprocess import Popen,PIPE,TimeoutExpired
from multiprocessing import Pool
from functools import partial
import numpy as np
import pandas as pd

sys.path.insert(1, 'swifitool') # use swifitool folder for file exports

from faults_inject import ExecConfig
from faults.jbe import JBE
from faults.jmp import JMP
from faults.z1b import Z1B
from faults.z1w import Z1W
from faults.nop import NOP
```

```
from faults.flp import FLP

class NoDaemonProcess(multiprocessing.Process):
    # make 'daemon' attribute always return False
    def _get_daemon(self):
        return False

def _set_daemon(self, value):
        pass
    daemon = property(_get_daemon, _set_daemon)

class MyPool(multiprocessing.pool.Pool):
    Process = NoDaemonProcess

def extract_x86_instructions(infile):
    print("Disassembling the binary and parsing instructions...\n");
    infile = open(infile, 'rb')
    # ELFFile looks for magic number, if there's none, ELFError is raised
    try:
        elffile = ELFFile(infile)
        parsing = False
        startAddress = 65535
        endAddress = 0
        # all jump instr supported by Intel x86 CPU
        supjumps = ['jne', 'je', 'jbe', 'jae', 'jb', 'jo', 'jmp', 'ja', 'jle',
'js', 'jc', 'jcxz', 'jecxz', 'jrcxz', 'jg', 'jge', 'jl', 'jle', 'jna', 'jnae',
'jnbe', 'jnc', 'jng', 'jnge', 'jnl', 'jnle', 'jno', 'jnp', 'jns', 'jnz', 'jp',
'jpe', 'jpo', 'jz']
        jumps = []  # array for jmp instructions
        cmpsmovs = []   # array for cmp and mov instructions
        allinstr = []   # all instructions' addresses and their size in bytes
        for section in elffile.iter_sections():
            ops = section.data()
            addr = section['sh_addr']
            name = section.name
            md = Cs(CS_ARCH_X86, CS_MODE_32)
            md.detail = True
            # print("%x\t%s\t%s" %(i.address, i.mnemonic, i.op_str))
            # print("%x:\t%s\t%s\t" %(i.address, i.mnemonic, i.op_str) +
              # ' '.join(format(x, '02x') for x in i.bytes)) # with bytes
            # below code finds and parses only certain elf sections
            # this is consistent with "objdump -S binary" command output
            if name == ".rodata": parsing = False
            elif name == ".init" or parsing:
                parsing = True
                for i in md.disasm(ops, addr):
                    # determine the heap range
                    if i.address < startAddress: startAddress=i.address
                    if i.address > endAddress: endAddress=i.address
                    allinstr.append({'addr':i.address, 'size':i.size})
                    # print("%x\t%s\t%s\t%d" %(i.address, i.mnemonic,
i.op_str, i.size))
                    # determine the instruction type and parse accordingly
                    if i.mnemonic in supjumps:  # select only jump
instructions
                        if len(i.op_str)==6: # process only simple jumps e.g
0x3eef
                            # print("%x\t%s\t%s" %(i.address, i.mnemonic,
i.op_str))
                            type = i.mnemonic
                            jumpfrom = hex(i.address) # 0xdead
```

```
                            jumpto = i.op_str   # 0xbeef
                            jump = {'type':type, 'from':jumpfrom, 'to':jumpto}
                            jumps.append(jump)
                    # zero static compare values and static variables
                    elif i.mnemonic == 'cmp' or i.mnemonic =='mov':  # select
cmp or mov instructions
                        # print("%x:\t%s\t%s\t%d" %(i.address, i.mnemonic,
i.op_str, i.size))
                        lastoperand = i.operands[len(i.operands)-1]
                        operands = i.op_str.split()
                        value = operands[len(operands)-1]
                        # ignore comparisons with zero and values stored in
registers
                        if value!='0' and ']' not in value and
lastoperand.type!=X86_OP_REG:
                            # print("%x:\t%s\t%s\t%d" %(i.address, i.mnemonic,
i.op_str, i.size))
                            size = 0
                            loc = 0
                            if len(value)<=4:   # '0x' + 1 byte i.e. max 255
                                size = 1
                                if 'byte' in operands:
                                    loc = hex(i.address + (i.size - 1))
                                elif 'word' in operands:
                                    loc = hex(i.address + (i.size - 2)) # 2
bytes
                                elif 'dword' in operands:
                                    loc = hex(i.address + (i.size - 4)) # 4
bytes
                            elif len(value)<=6: # '0x' + 2 bytes i.e. uint16_t
                                size = 2
                                if 'word' in operands:
                                    loc = hex(i.address + (i.size - 2)) # 2
bytes
                                elif 'dword' in operands:
                                    loc = hex(i.address + (i.size - 4)) # 4
bytes
                            elif len(value)<=10: # '0x' + 4 bytes i.e.
uint32_t or int
                                size = 4
                                if 'dword' in operands:
                                    loc = hex(i.address + (i.size - 4)) # 4
bytes
                            if loc!=0:

cmpsmovs.append({'type':i.mnemonic,'size':size,'loc':loc})
        return allinstr, jumps, cmpsmovs
    except ELFError:
        logging.info("%s is invalid elf file" % elffile)

def extract_arm_instructions(infile):
    print("Disassembling the binary and parsing instructions...\n");
    infile = open(infile, 'rb')
    # ELFFile looks for magic number, if there's none, ELFError is raised
    try:
        elffile = ELFFile(infile)
        parsing = False
        startAddress = 65535
        endAddress = 0
        # all ARM branch instructions
```

```
        branch_instr = ['b', 'beq', 'bne', 'bcs', 'bhs', 'bcc', 'blo',
'bmi','bpl',
        'bvs', 'bvc', 'bhi', 'bls', 'bge', 'blt', 'bgt', 'ble', 'bl', 'bleq',
        'bllt', 'blx', 'bx', 'bxeq', 'bxne', 'bxcs', 'bxcc', 'bxhi', 'bxls',
        'bxgt', 'bxle']
        jumps = []  # array for jmp instructions
        cmpsmovs = []   # array for cmp and mov instructions
        allinstr = []   # all instructions' addresses and their size in bytes
        for section in elffile.iter_sections():
            ops = section.data()
            addr = section['sh_addr']   # section start address
            offset = section['sh_offset']
            file_offset = addr - offset
            name = section.name
            md = Cs(CS_ARCH_ARM, CS_MODE_ARM)
            # below code finds and parses only certain elf sections
            # this is consistent with "objdump -S binary" command output
            if name == ".rodata": parsing = False
            elif name == ".init" or parsing:
                parsing = True
                for i in md.disasm(ops, addr):
                    # determine the heap range
                    if i.address < startAddress: startAddress=i.address
                    if i.address > endAddress: endAddress=i.address
                    allinstr.append({'addr':i.address-file_offset,
'size':i.size})
                    # determine the instruction type and parse accordingly
                    if i.mnemonic in branch_instr:  # select only branch
instructions
                        if len(i.op_str)>4: # process proper jump addresses
and ignore registers
                            type = i.mnemonic
                            jumpfrom = hex(i.address-file_offset)
                            jumpto = hex(int(i.op_str.split('#')[1],0)-
file_offset) # remove # from '#0x14f30'
                            jump = {'type':type, 'from':jumpfrom, 'to':jumpto}
                            jumps.append(jump)
                    # zero static compare values and static variables
                    elif i.mnemonic == 'cmp' or i.mnemonic =='mov':  # select
cmp or mov instructions
                        operands = i.op_str.split()
                        op_value = operands[len(operands)-1]
                        # ignore comparisons with zero and values stored in
registers
                        if op_value!='#0' and '#' in op_value:
                            val = op_value.split('#')[1]
                            loc = hex(i.address)
                            if len(val)<=4:   # '0x' + 1 byte i.e. max 255
                                size = 1
                            elif len(val)<=6: # '0x' + 2 bytes
                                size = 2

cmpsmovs.append({'type':i.mnemonic,'size':size,'loc':loc})

        print("Number of jumps: ", len(jumps))
        print("Number of cmpsmovs: ", len(cmpsmovs))
        return allinstr, jumps, cmpsmovs
    except ELFError:
        logging.info("%s is invalid elf file" % elffile)

def copy_original_file(infile):
```

```
    print(infile)
    Path("%s-none-faulted-binaries" %(infile)).mkdir(parents=True,
exist_ok=True)
    outfile = "%s-none-faulted-binaries/" %(infile)
    shutil.copy(infile,outfile)

def inject_jump_faults(jumps,allinstr,infile,arch):
    # General configuration
    config = ExecConfig(os.path.expanduser(infile), None, arch, None) # None
for outfile and wordsize
    # prepare the fault models
    fm_list = []
    jump_targets = [j['to'] for j in jumps]
    jump_targets = list(dict.fromkeys(jump_targets)) # remove duplicates
    #print("The jump_targets array is: ", jump_targets)
    for idx,jump in enumerate(jumps):
        for target in allinstr:
            if target['addr']!=jump['to']:
                try:
                    for offset in range(0,target['size']):
                        loc = hex(target['addr']+offset)
                        if jump['type'] == ('jmp' or 'b'):
                            if offset>0:
                                type = jump['type'] + '_middlejmp'
                                print(type)
                                fault = {'type':type,'at':jump['from'],
                                    'from':jump['to'],'to':loc,
                                    'fault':JMP(config,[jump['from'],loc])}
                            else:
                                fault =
{'type':jump['type'],'at':jump['from'],
                                    'from':jump['to'],'to':loc,
                                    'fault':JMP(config,[jump['from'],loc])}
                        else:
                            if offset>0:
                                type = jump['type'] + '_middlejmp'
                                fault = {'type':type,'at':jump['from'],
                                    'from':jump['to'],'to':loc,
                                    'fault':JBE(config,[jump['from'],loc])}
                            else:
                                fault =
{'type':jump['type'],'at':jump['from'],
                                    'from':jump['to'],'to':loc,
                                    'fault':JBE(config, [jump['from'],loc])}
                        fm_list.append(fault)
                except SystemExit:
                    pass # skip targets causing out of range erors and move on

    print("Number of new binaries with changed jumps: ", len(fm_list))
    # create a folder for faulted binaries
    Path("%s-jmp-faulted-binaries" %(infile)).mkdir(parents=True,
exist_ok=True)
    # Duplicate the input and then apply the faults
    for idx,f in enumerate(fm_list):
        print("Creating jump fault binary ", idx)
        outfile = '%s-jmp-faulted-binaries/%s_at_%s_from_%s_to_%s' %(infile,
f['type'],
        f['at'],f['from'],f['to'])
        shutil.copy(infile,outfile)
        with open(outfile, "r+b") as file:
            f['fault'].apply(file)
```

```
def inject_zero_faults(targets,infile,arch):
    # prepare the fault models
    fm_list = []
    for idx,target in enumerate(targets):
        try:
            if target['size'] == 1:
                config = ExecConfig(os.path.expanduser(infile), None, arch,
None) # None for outfile and wordsize
                fault = {'type':target['type'], 'loc':target['loc'],
'fault':Z1B(config,[target['loc']])}
                fm_list.append(fault)
            else:
                config = ExecConfig(os.path.expanduser(infile), None, arch,
target['size'])
                fault = {'type':target['type'], 'loc':target['loc'],
'fault':Z1W(config,[target['loc']])}
                fm_list.append(fault)
        except SystemExit:
            pass # skip targets causing out of range erors and move on
    # print("Number of locations to zero: ", len(targets))
    print("Number of new binaries with zeroed values: ", len(fm_list))
    # create a folder for faulted binaries
    Path("%s-zero-faulted-binaries" %(infile)).mkdir(parents=True,
exist_ok=True)
    # Duplicate the input and then apply the faults
    for idx,f in enumerate(fm_list):
        print("Creating zero fault binary ", idx)
        outfile = '%s-zero-faulted-binaries/%s_at_%s_zeroed' %(infile,
f['type'],f['loc'])
        shutil.copy(infile,outfile)
        with open(outfile, "r+b") as file:
            f['fault'].apply(file)

def inject_nop_faults(targets, infile, arch):
    # prepare the fault models
    fm_list = []
    for idx,target in enumerate(targets):
        try:
            config = ExecConfig(os.path.expanduser(infile), None, arch, None)
# None for outfile and wordsize
            addr_from = target['addr']
            addr_till = target['addr'] + target['size'] - 1
            noprange = hex(addr_from) + '-' + hex(addr_till)
            print("From %x till %x" %(addr_from,addr_till))
            #print("noprange: ", noprange)
            fault = {'range':noprange, 'fault':NOP(config,[noprange])}
            fm_list.append(fault)
        except SystemExit:
            pass # skip targets causing out of range erors and move on
    # print("Number of instructions to be NOPed: ", len(targets))
    print("Number of new binaries with NOPed instructions: ", len(fm_list))
    # create a folder for faulted binaries
    Path("%s-nop-faulted-binaries" %(infile)).mkdir(parents=True,
exist_ok=True)
    # Duplicate the input and then apply the faults
    for idx,f in enumerate(fm_list):
        print("Creating NOP fault binary ", idx)
        outfile = '%s-nop-faulted-binaries/nop_%s' %(infile, f['range'])
        shutil.copy(infile,outfile)
        with open(outfile, "r+b") as file:
```

```
            f['fault'].apply(file)

def inject_flp_faults(targets, infile, arch):
    # prepare the fault models
    fm_list = []
    for idx,target in enumerate(targets):
        try:
            config = ExecConfig(os.path.expanduser(infile), None, arch, None)
# None for outfile and wordsize
            addr_from = target['addr']
            for offset in range(0,target['size']):
                loc = hex(addr_from+offset)

                #with static significance bit
                sgnf = random.randint(0,7)
                fault = {'loc':loc, 'sgnf':sgnf,
'fault':FLP(config,[loc,sgnf])}
                fm_list.append(fault)

                # or with varied significance bit
                #for sgnf in range(0,5):
                 #   fault = {'loc':loc, 'sgnf':sgnf,
'fault':FLP(config,[loc,sgnf])}
                  #  fm_list.append(fault)
        except SystemExit:
            pass # skip targets causing out of range erors and move on
    # print("Number of instructions to be FLPed: ", len(targets))
    print("Number of new binaries with FLPed instructions: ", len(fm_list))
    # create a folder for faulted binaries
    Path("%s-flp-faulted-binaries" %(infile)).mkdir(parents=True,
exist_ok=True)
    # Duplicate the input and then apply the faults
    for idx,f in enumerate(fm_list):
        try:
            print("Creating flip fault binary ", idx)
            outfile = '%s-flp-faulted-binaries/flp_at_%s_sgnf_%d' %(infile,
f['loc'],f['sgnf'])
            shutil.copy(infile,outfile)
            with open(outfile, "r+b") as file:
                f['fault'].apply(file)
        except:
            continue

def execute_with_each_input_nova(infile, arch, fault_type, batchsize,
faulty_binaries_list, row):
    print("Testing for temperature %s and light %s with index %s" %(row[1],
row[2], row[0]))
    func = partial(execute_file_nova, "%s-%s-faulted-binaries" %(infile,
fault_type), arch, row[1], row[2])
    execute_with_each_input(infile, fault_type, batchsize,
faulty_binaries_list, func, row)

def execute_with_each_input_verifypin(infile, arch, fault_type, batchsize,
faulty_binaries_list, row):
    print("Testing for card_pin %s and user_pin %s with index %s" %(row[1],
row[2], row[0]))
    func = partial(execute_file_verifypin, "%s-%s-faulted-binaries" %(infile,
fault_type), arch, row[1], row[2])
    execute_with_each_input(infile, fault_type, batchsize,
faulty_binaries_list, func, row)
```

```
def execute_with_each_input(infile, fault_type, batchsize,
faulty_binaries_list, func, row):
    for i in range(0, len(faulty_binaries_list), batchsize):
        print("Executing binaries ", i, " : ", i+batchsize)
        batch = faulty_binaries_list[i:i+batchsize]
        with Pool() as pool:
            results = pool.imap(func, batch)
            pool.close()
            try:
                with open("%s_%s_results.csv" %(infile, fault_type), 'a') as
csvfile:
                    writer = csv.writer(csvfile, delimiter=',')
                    for idx,res in enumerate(results):
                        writer.writerow([infile, res['filename'], row[0],
row[1], row[2], res['stdout'],
                                         res['stderr'], res['exitcode'],
res['timedout'], fault_type.upper()])
            except:
                traceback.print_exc()
                continue

def execute_random_file_with_each_input_nova(infile, arch, fault_type, batch,
row):
    print("Testing for temperature %s and light %s with index %s" %(row[1],
row[2], row[0]))
    func = partial(execute_file_nova, "%s-%s-faulted-binaries" %(infile,
fault_type), arch, row[1], row[2])
    execute_random_file_with_each_input(infile, fault_type, batch, func, row)

def execute_random_file_with_each_input_verifypin(infile, arch, fault_type,
batch, row):
    print("Testing for card_pin %s and user_pin %s with index %s" %(row[1],
row[2], row[0]))
    func = partial(execute_file_verifypin, "%s-%s-faulted-binaries" %(infile,
fault_type), arch, row[1], row[2])
    execute_random_file_with_each_input(infile, fault_type, batch, func, row)

def execute_random_file_with_each_input(infile, fault_type, batch, func, row):
    with Pool(processes=1) as pool:
        results = pool.imap(func, batch)
        pool.close()
        try:
             with open("%s_%s_results.csv" %(infile, fault_type), 'a') as
csvfile:
                writer = csv.writer(csvfile, delimiter=',')
                for idx,res in enumerate(results):
                    writer.writerow([infile, res['filename'], row[0], row[1],
row[2], res['stdout'],
                                     res['stderr'], res['exitcode'],
res['timedout'], fault_type.upper()])
        except:
            traceback.print_exc()

def execute_file_nova(dirname, arch, temperature, light, filename):
    if arch=='x86':
        command = '%s/%s %s %s' %(dirname, filename, temperature, light)
    elif arch=='arm':
        command = 'qemu-arm -L /usr/arm-linux-gnueabi/ %s/%s %s %s' %(dirname,
filename, temperature, light)
    return execute_file(command, filename)
```

```
def execute_file_verifypin(dirname, arch, card_pin, user_pin, filename):
    if arch=='x86':
        command = '%s/%s %s %s' %(dirname, filename, card_pin, user_pin)
    elif arch=='arm':
        command = 'qemu-arm -L /usr/arm-linux-gnueabi/ %s/%s %s %s' %(dirname,
filename, card_pin, user_pin)
    return execute_file(command, filename)

def execute_file(command, filename):
    args = shlex.split(command)
    # p = Popen(args,stdout=PIPE,stderr=PIPE,universal_newlines=True) #
extract stdout in a textual utf-8 format
    p = Popen(args,stdout=PIPE,stderr=PIPE) # extract stdout in a binary-like
format
    try:
        outs, errs = p.communicate(timeout=3)   # 3 sec
        # print(filename,outs,errs,p.returncode)
        return({'filename':filename,'stdout':outs,'stderr':errs,
            'exitcode':p.returncode,'timedout':False})
    except TimeoutExpired:
        p.kill()
        outs, errs = p.communicate()
        # print(filename,outs,errs,p.returncode)
        return({'filename':filename,'stdout':outs,'stderr':errs,
            'exitcode':p.returncode,'timedout':True})
    finally:
        p.kill()

def run_all_faulty_executables_random_input_nova(infile, arch, fault_type,
input_data, batchsize=50):
    print("\nRunning the faulty binaries and recording the results...\n")
    print("This may take a while...\n")
    with open("%s_%s_results.csv" %(infile, fault_type), 'w') as csvfile:
        writer = csv.writer(csvfile, delimiter=',')
        writer.writerow(['File_Name', 'Faulty_Executable', 'Og_Data_Index',
'Temperature', 'Light', 'Output',
                         'Error', 'Exit_Code', 'Time_Out', 'Fault_Type'])
    faulty_binaries_list = os.listdir("%s-%s-faulted-binaries" %(infile,
fault_type))
    print("Total binaries to execute: ", len(faulty_binaries_list))
    with open("%s_%s_results.csv" %(infile, fault_type), 'a') as csvfile:
        writer = csv.writer(csvfile, delimiter=',')
        for i in range(0, len(faulty_binaries_list), batchsize):
            print("Executing binaries ", i, " : ", i+batchsize)
            batch = faulty_binaries_list[i:i+batchsize]
            j = random.randint(0, (input_data.shape[0])-1)
            temperature = input_data.iloc[j].temperature
            light = input_data.iloc[j].light
            print("Testing for temperature %s and light %s with index %s"
%(temperature, light, i))
            func = partial(execute_file_nova, "%s-%s-faulted-binaries"
%(infile, fault_type), arch, temperature, light)
            with Pool(processes=batchsize) as pool:
                results = pool.imap(func, batch)
                pool.close()
                for idx,res in enumerate(results):
                    writer.writerow([infile, res['filename'], i, temperature,
light, res['stdout'],
                                     res['stderr'], res['exitcode'],
res['timedout'], fault_type.upper()])
```

```
def run_faulty_executables_nova(infile, arch, fault_type, input_data,
batchsize=50):
    print("\nRunning the faulty binaries and recording the results...\n")
    print("This may take a while...\n")
    with open("%s_%s_results.csv" %(infile, fault_type), 'w') as csvfile:
        writer = csv.writer(csvfile, delimiter=',')
        writer.writerow(['File_Name', 'Faulty_Executable', 'Og_Data_Index',
'Temperature', 'Light', 'Output',
                         'Error', 'Exit_Code', 'Time_Out', 'Fault_Type'])
    faulty_binaries_list = os.listdir("%s-%s-faulted-binaries" %(infile,
fault_type))
    print("Total binaries to execute: ", len(faulty_binaries_list))
    with open("%s_%s_results.csv" %(infile, fault_type), 'a') as csvfile:
        writer = csv.writer(csvfile, delimiter=',')
        for index, row in input_data.iterrows():
            print("Testing for temperature %s and light %s with index %s"
%(row['temperature'], row['light'], index))
            func = partial(execute_file_nova, "%s-%s-faulted-binaries"
%(infile, fault_type), arch, row['temperature'], row['light'])
            for i in range(0, len(faulty_binaries_list), batchsize):
                print("Executing binaries ", i, " : ", i+batchsize)
                batch = faulty_binaries_list[i:i+batchsize]
                with Pool(processes=batchsize) as pool:
                    results = pool.imap(func, batch)
                    pool.close()
                    for idx,res in enumerate(results):
                        writer.writerow([infile, res['filename'], index,
row['temperature'], row['light'], res['stdout'],
                                         res['stderr'], res['exitcode'],
res['timedout'], fault_type.upper()])

def run_faulty_executables_verifypin(infile, arch, fault_type, input_data,
batchsize=50):
    print("\nRunning the faulty binaries and recording the results...\n")
    print("This may take a while...\n")
    with open("%s_%s_results.csv" %(infile, fault_type), 'w') as csvfile:
        writer = csv.writer(csvfile, delimiter=',')
        writer.writerow(['File_Name', 'Faulty_Executable', 'Og_Data_Index',
'Card_Pin', 'User_Pin', 'Output',
                         'Error', 'Exit_Code', 'Time_Out', 'Fault_Type'])
    faulty_binaries_list = os.listdir("%s-%s-faulted-binaries" %(infile,
fault_type))
    print("Total binaries to execute: ", len(faulty_binaries_list))
    with open("%s_%s_results.csv" %(infile, fault_type), 'a') as csvfile:
        writer = csv.writer(csvfile, delimiter=',')
        for index, row in input_data.iterrows():
            print("Testing for card_pin %s and user_pin %s with index %s"
%(row['card_pin'], row['user_pin'], index))
            func = partial(execute_file_verifypin, "%s-%s-faulted-binaries"
%(infile, fault_type), arch, row['card_pin'], row['user_pin'])
            for i in range(0, len(faulty_binaries_list), batchsize):
                print("Executing binaries ", i, " : ", i+batchsize)
                batch = faulty_binaries_list[i:i+batchsize]
                with Pool(processes=batchsize) as pool:
                    results = pool.imap(func, batch)
                    pool.close()
                    for idx,res in enumerate(results):
                        writer.writerow([infile, res['filename'], index,
row['card_pin'], row['user_pin'], res['stdout'],
                                         res['stderr'], res['exitcode'],
res['timedout'], fault_type.upper()])
```

```
def run_fixed_faulty_executables_nova(infile, arch, fault_type, input_data,
faulty_binaries_list, batchsize=50):
    print("\nRunning the faulty binaries and recording the results...\n")
    print("This may take a while...\n")
    with open("%s_%s_results.csv" %(infile, fault_type), 'w') as csvfile:
        writer = csv.writer(csvfile, delimiter=',')
        writer.writerow(['File_Name', 'Faulty_Executable', 'Og_Data_Index',
'Temperature', 'Light', 'Output',
                         'Error', 'Exit_Code', 'Time_Out', 'Fault_Type'])
    print("Total binaries: ", len(faulty_binaries_list))
    input_batch_size = int(input_data.shape[0]/50)
    for i in range(0, len(faulty_binaries_list), batchsize):
        for j in range(0, input_data.shape[0], input_batch_size):
            print("Executing inputs ", j, " : ", input_batch_size)
            input_batch = input_data[j : j+input_batch_size].to_records()
            try:
                with MyPool(processes=50) as input_pool:
                    print("Executing binaries ", i, " : ", i+batchsize)
                    batch = faulty_binaries_list[i:i+batchsize]
                    func_execute_with_each_input =
partial(execute_random_file_with_each_input_nova, infile, arch, fault_type,
batch)
                    input_results =
input_pool.imap(func_execute_with_each_input, input_batch)
                    input_pool.close()
                    list(enumerate(input_results))
            except:
                traceback.print_exc()
                continue

def run_fixed_faulty_executables_verifypin(infile, arch, fault_type,
input_data, faulty_binaries_list, batchsize=50):
    print("\nRunning the faulty binaries and recording the results...\n")
    print("This may take a while...\n")
    with open("%s_%s_results.csv" %(infile, fault_type), 'w') as csvfile:
        writer = csv.writer(csvfile, delimiter=',')
        writer.writerow(['File_Name', 'Faulty_Executable', 'Og_Data_Index',
'Card_Pin', 'User_Pin', 'Output',
                         'Error', 'Exit_Code', 'Time_Out', 'Fault_Type'])
    print("Total binaries: ", len(faulty_binaries_list))
    input_batch_size = int(input_data.shape[0]/50)
    for i in range(0, len(faulty_binaries_list), batchsize):
        for j in range(0, input_data.shape[0], input_batch_size):
            print("Executing inputs ", j, " : ", input_batch_size)
            input_batch = input_data[j : j+input_batch_size].to_records()
            try:
                with MyPool(processes=50) as input_pool:
                    print("Executing binaries ", i, " : ", i+batchsize)
                    batch = faulty_binaries_list[i:i+batchsize]
                    func_execute_with_each_input =
partial(execute_random_file_with_each_input_verifypin, infile, arch,
fault_type, batch)
                    input_results =
input_pool.imap(func_execute_with_each_input, input_batch)
                    input_pool.close()
                    list(enumerate(input_results))
            except:
                traceback.print_exc()
                continue
```

```
def run_random_faulty_executables_nova(infile, arch, fault_type, input_data,
batchsize=50):
    print("\nRunning the faulty binaries and recording the results...\n")
    print("This may take a while...\n")
    with open("%s_%s_results.csv" %(infile, fault_type), 'w') as csvfile:
        writer = csv.writer(csvfile, delimiter=',')
        writer.writerow(['File_Name', 'Faulty_Executable', 'Og_Data_Index',
'Temperature', 'Light', 'Output',
                         'Error', 'Exit_Code', 'Time_Out', 'Fault_Type'])
    faulty_binaries_list = os.listdir("%s-%s-faulted-binaries" %(infile,
fault_type))
    print("Total binaries: ", len(faulty_binaries_list))
    input_batch_size = int(input_data.shape[0]/50)
    for j in range(0, input_data.shape[0], input_batch_size):
        print("Executing inputs ", j, " : ", input_batch_size)
        input_batch = input_data[j : j+input_batch_size].to_records()
        try:
            with MyPool(processes=50) as input_pool:
                i = random.randint(0, len(faulty_binaries_list)-1)
                print("Executing binaries ", i, " : ", i+batchsize)
                batch = faulty_binaries_list[i:i+batchsize]
                func_execute_with_each_input =
partial(execute_random_file_with_each_input_nova, infile, arch, fault_type,
batch)
                input_results = input_pool.imap(func_execute_with_each_input,
input_batch)
                input_pool.close()
                list(enumerate(input_results))
            map(lambda x: faulty_binaries_list.remove(x), batch)
        except:
            traceback.print_exc()
            continue

def run_random_faulty_executables_verifypin(infile, arch, fault_type,
input_data, batchsize=50):
    print("\nRunning the faulty binaries and recording the results...\n")
    print("This may take a while...\n")
    with open("%s_%s_results.csv" %(infile, fault_type), 'w') as csvfile:
        writer = csv.writer(csvfile, delimiter=',')
        writer.writerow(['File_Name', 'Faulty_Executable', 'Og_Data_Index',
'Card_Pin', 'User_Pin', 'Output',
                         'Error', 'Exit_Code', 'Time_Out', 'Fault_Type'])
    faulty_binaries_list = os.listdir("%s-%s-faulted-binaries" %(infile,
fault_type))
    print("Total binaries: ", len(faulty_binaries_list))
    input_batch_size = int(input_data.shape[0]/5)
    for j in range(0, input_data.shape[0], input_batch_size):
        print("Executing inputs ", j, " : ", input_batch_size)
        input_batch = input_data[j : j+input_batch_size].to_records()
        try:
            with MyPool(processes=50) as input_pool:
                i = random.randint(0, len(faulty_binaries_list)-1)
                print("Executing binaries ", i, " : ", i+batchsize)
                batch = faulty_binaries_list[i:i+batchsize]
                func_execute_with_each_input =
partial(execute_random_file_with_each_input_verifypin, infile, arch,
fault_type, batch)
                input_results = input_pool.imap(func_execute_with_each_input,
input_batch)
                input_pool.close()
                list(enumerate(input_results))
```

```
            map(lambda x: faulty_binaries_list.remove(x), batch)
        except:
            traceback.print_exc()
            continue

def main(argv):

    CLI=argparse.ArgumentParser()
    CLI.add_argument(
        "--filename",
        type=str,
        default='NovaHomeDaemon_Ext'
    )
    CLI.add_argument(
        "--arch",
        type=str,
        default='arm'
    )
    CLI.add_argument(
        "--fault_list",
        nargs="*",
        type=str,
        default=['none']
    )


    args = CLI.parse_args()
    infile = args.filename
    arch = args.arch
    fault_types = args.fault_list


    if arch=='x86':
        allinstr, jumps, cmpsmovs = extract_x86_instructions(infile)
    elif arch=='arm':
        allinstr, jumps, cmpsmovs = extract_arm_instructions(infile)
    print("Number of detected instructions: ", len(allinstr))

    # Nova Smart Home Control Daemon
    for fault_type in fault_types:
        print("fault_type: ", fault_type)
        if(fault_type=='none'):
            copy_original_file(infile)
            print("Reading Data....")
            col_list = ['temperature', 'light']
            input_data = pd.read_excel("./Nova_Input_Data.xlsx",
sheet_name='Input_Data', usecols=col_list)
            input_data = input_data.dropna()
            run_faulty_executables_nova(infile, arch, fault_type, input_data,
batch_size=1)
        if(fault_type=='jmp'):
            inject_jump_faults(jumps,allinstr,infile,arch)
            print("Reading Data....")
            col_list = ['temperature', 'light']
            input_data = pd.read_excel("./Nova_Input_Data.xlsx",
sheet_name='Input_Data', usecols=col_list)
            input_data = input_data.dropna()
            run_all_faulty_executables_random_input_nova(infile, arch,
fault_type, input_data)
        if(fault_type=='zero'):
            inject_zero_faults(cmpsmovs,infile,arch)
```

```
            print("Reading Data....")
            col_list = ['temperature', 'light']
            input_data = pd.read_excel("./Nova_Input_Data.xlsx",
sheet_name='Input_Data', usecols=col_list)
            input_data = input_data.dropna()
            run_all_faulty_executables_random_input_nova(infile, arch,
fault_type, input_data)
        if(fault_type=='nop'):
            inject_nop_faults(allinstr,infile,arch)
            print("Reading Data....")
            col_list = ['temperature', 'light']
            input_data = pd.read_excel("./Nova_Input_Data.xlsx",
sheet_name='Input_Data', usecols=col_list)
            input_data = input_data.dropna()
            run_all_faulty_executables_random_input_nova(infile, arch,
fault_type, input_data)
        if(fault_type=='flp'):
            inject_flp_faults(allinstr,infile,arch)
            print("Reading Data....")
            col_list = ['temperature', 'light']
            input_data = pd.read_excel("./Nova_Input_Data.xlsx",
sheet_name='Input_Data', usecols=col_list)
            input_data = input_data.dropna()
            '''run_faulty_executables(infile, arch, fault_type, input_data)
            input_data = input_data[0:40000]
            run_fixed_faulty_executables(infile, arch, fault_type, input_data,
['flp_at_0x2c8_sgnf_7', 'flp_at_0x1c1c_sgnf_3', 'flp_at_0x1ea8_sgnf_1',

'flp_at_0x1eac_sgnf_4', 'flp_at_0x1b54_sgnf_7'])'''
            run_all_faulty_executables_random_input_nova(infile, arch,
fault_type, input_data)



    # VerifyPIN
    for fault_type in fault_types:
        print("fault_type: ", fault_type)
        if(fault_type=='none'):
            copy_original_file(infile)
            print("Reading Data....")
            col_list = ['card_pin', 'user_pin']
            input_data = pd.read_excel("./VerifyPin_Input_Data.xlsx",
sheet_name='Input_Data', usecols=col_list)
            input_data = input_data.dropna()
            run_faulty_executables_verifypin(infile, arch, fault_type,
input_data, batch_size=1)
        if(fault_type=='jmp'):
            inject_jump_faults(jumps,allinstr,infile,arch)
            print("Reading Data....")
            col_list = ['card_pin', 'user_pin']
            input_data = pd.read_excel("./VerifyPin_Input_Data.xlsx",
sheet_name='Input_Data', usecols=col_list)
            input_data = input_data.dropna()
            run_random_faulty_executables_verifypin(infile, arch, fault_type,
input_data)
        if(fault_type=='zero'):
            inject_zero_faults(cmpsmovs,infile,arch)
            print("Reading Data....")
            col_list = ['card_pin', 'user_pin']
            input_data = pd.read_excel("./VerifyPin_Input_Data.xlsx",
sheet_name='Input_Data', usecols=col_list)
```

```
            input_data = input_data.dropna()
            run_random_faulty_executables_verifypin(infile, arch, fault_type,
input_data)
        if(fault_type=='nop'):
            inject_nop_faults(allinstr,infile,arch)
            print("Reading Data....")
            col_list = ['card_pin', 'user_pin']
            input_data = pd.read_excel("./VerifyPin_Input_Data.xlsx",
sheet_name='Input_Data', usecols=col_list)
            input_data = input_data.dropna()
            run_random_faulty_executables_verifypin(infile, arch, fault_type,
input_data)
        if(fault_type=='flp'):
            inject_flp_faults(allinstr,infile,arch)
            print("Reading Data....")
            col_list = ['card_pin', 'user_pin']
            input_data = pd.read_excel("./VerifyPin_Input_Data.xlsx",
sheet_name='Input_Data', usecols=col_list)
            input_data = input_data.dropna()
            input_data = input_data[0:40000]
            run_fixed_faulty_executables_verifypin(infile, arch, fault_type,
input_data, ['flp_at_0xea2_sgnf_6', 'flp_at_0xca2_sgnf_4',
'flp_at_0xdda_sgnf_7', 'flp_at_0xeac_sgnf_7', 'flp_at_0xe39_sgnf_6'])


if __name__ == '__main__':
    main(sys.argv)
```

Appendix B. Case Studies

The sources of the VerifyPIN and Nova Smart Home software utilized in the case studies are provided below:

VerifyPIN – https://lazart.gricad-pages.univ-grenoble-alpes.fr/fissc/

Nova Smart Home – https://github.com/SIN0VA/Nova-Smart-Home