Finding Software Vulnerabilities in Unmanned Aerial Vehicles

A dissertation submitted to the University of Manchester for the degree of Master of Science in the Faculty of Engineering and Physical Sciences

Samih R. Alkhamissi

School of Computer Science

2019

ABSTRACT

As technology advances at an increasingly accelerated pace, validation programmes for technological applications must be at the heart of new technologies. The results of these programs influence the performance and development of robust and durable technological applications in the future. With the presence of development and expansion attack strategies caused by low-level and high-level vulnerabilities in these applications, security performs a significant role. Sufficient software verification and testing is vital to guarantee the safety and reliability of applications. An important characteristic of software reliability is a safety guarantee which refers to the complete absence of low-level vulnerabilities that could cause exploitation. An early software crash indicates possible software failures such as memory corruption or zero division. The presence of safety or security-critical failures can be indicated by such accidents.

As verification techniques are still required to develop, expand and implement integration techniques, this dissertation examines several of these aspects of development. This dissertation developed the Depthk 3.2 project using a framework for integrating static and dynamic analysis techniques for UAV software written by C++. The main objective of this project is to detect and prevent safety vulnerabilities caused by low-level vulnerabilities. The first step of the tool is source code analysis by abstract interpretation as a static analysis technique. The final step is fuzz testing as a dynamic analysis technique to avoid attacks of incorrect and unwanted inputs and to reduce false positives resulting from static analysis. Experimental results indicate the theoretical strength of the idea however, due to time limitations, there were some practical limitations resulting from the configuration of the Depthk 3.1.

Contents

0.1 List of Abbreviations vii
0.2 Declaration viii
0.3 Intellectual property statement ix
0.4 Acknowledgments x

1 INTRODUCTION 1
1.1 The Problem Statement 3
1.2 Outline of the Solution 3
1.3 Research Questions 3
1.4 Aim and Objectives 4
1.5 Research Methodology 4
1.6 Project Deliverables 5
1.7 Dissertation Structure 5

2 PRELIMINARIES 8
2.1 Model Checking 8
2.2 Bounded Model Checking of Software 9
2.3 ESBMC 10
2.3.1 Features 11
2.4 Abstract Interpretation 12
2.5 Fuzz Testing 14
2.6 Vulnerabilities Overview 15
2.6.1 Buffer Overflow 16
2.6.2 Integer Overflow 18
2.6.3 Division By Zero 19
2.6.4 NULL Pointer Dereference 20

3 SYSTEMATIC LITERATURE REVIEW 21
3.1 Literature Review Objectives 21
3.2 Search Strategy 21

3.2.1 Data Resources 22
3.3 Study Selection 23
3.4 Study Quality Assessment 23
3.5 Search Results 24
3.6 Review of Software Vulnerability Detection's Studies 24
3.6.1 Bounded Model Checking Verification 25
3.6.2 Abstract Interpretation Verification 26
3.6.3 Verification by Fuzz Testing 27
3.6.4 Hybrid Verification 28
3.6.5 Cutting-Edge Software Verification Tools 29
3.6.6 Summary and Conclusion 31

4 METHOD AND RESEARCH DESIGN 32
4.1 Project Rationale 32
4.2 Development Methodology 32
4.2.1 V-Model Technique 33
4.3 DepthK 3.2 Architecture 34
4.4 Architecture Internal Details 37
4.4.1 Fuzz Testing Agent 37
4.4.2 Abstract Interpretation Agent 38
4.4.3 Bounded Model Checker Agent 41
4.4.4 Benchmark Selected 42

5 IMPLEMENTATION 44
5.1 Implementation Plan 44
5.1.1 Configuration Environment 45
5.2 The Back-End Structure 46
5.2.1 Implementing Abstract Interpretation and Bounded Model Checker . . 46
5.2.2 Implementing Fuzz Testing Agent 49
5.3 Display output 52
5.3.1 Graphical User Interface 52
5.4 Implementation Challenges 53

6 EVALUATION AND ANALYSIS OF DIFFERENT VERIFICATION TOOL 54
6.1 Comparing the accuracy and performance of verification tools 54
6.1.1 Evaluation Metrics 55
6.1.2 Evaluation Result 56
6.1.3 Security Vulnerability coverage 57
6.2 Performance Evaluation and Discussion 57

7 CONCLUSIONS **59**
7.1 Conclusions Summary 59
7.2 Limitations 60
7.3 Future Work 60

Bibliography **61**

List of Tables

1.1 Recent Losses in Aerospace due to Software Error by David et al. [1] 2

2.1 Comparison of Fuzzer Based on Generation and Mutations by [2] 14

3.1 Keywords and Search Terms 22
3.2 Checklist for Resource Quality Assessment 24
3.3 Digital Library Search Results Using Search Strings 24
3.4 Summary of Cutting-Edge Software Verification Tools 30
3.5 Comparison of State-of the-Art Techniques 31

5.1 Development Environment Tools 45
5.2 Some Example of Pycparsers' Modules 47
5.3 Fuzz' Modules by FuzzingLib 50

6.1 Open-Source verification Tools 55
6.2 Time and Warnings Generated by Each Tool 56
6.3 Warning Counts for the Software Verification Tools 57
6.4 Coverage Comparison of DepthK 3.2 With Other Tools 57
6.5 Performance Evaluation for Open-Source Verification Tools 57

List of Figures

1.1 The Design-Science Approach to Research, the Technology Research Approach, and Dissertation's Approach . . . 6

2.1 The Model-Checking Approach . . . 9
2.2 Bounded Model Checking Approach by [3] . . . 10
2.3 ESBMC Architecture by [4] . . . 12
2.4 An Example of a C++ Code and its CFG [5] . . . 13
2.5 The Multiple Sorts of Fuzzing by [6] . . . 15
2.6 Sampling of Security Incidents by Attack Type, Time and Impact, 2014-2016 by [7] . . . 16
2.7 A Buffer Overflow Example Code . . . 17
2.8 Top Attack Types for Monitored Security by [7] . . . 18
2.9 An Example Code for Integer Overflow by [8] . . . 19
2.10 A Divide By Zero Code by [9] . . . 20
2.11 Possible NULL Pointer Dereference Vulnerability by [10] . . . 20

4.1 Software Development Life Cycle . . . 33
4.2 High-Level Architecture of UAV's Vulnerability Security Verification Tool. . . 35
4.3 Fuzz Agent Architecture . . . 37
4.4 Low-Level Architecture of Abstract Interpretation. . . . 39
4.5 Clang's Compiler Architecture . . . 39
4.6 Low-Level Of Bounded Model Checker Architecture . . . 41

5.1 An Example Code for pycparser Module . . . 47
5.2 Singleton's Logging . . . 49
5.3 Test Random Data Generation . . . 50
5.4 Execution Class and its Function . . . 51
5.5 Run Status Values . . . 51
5.6 User-Interface for the Prototype . . . 52

5.7 User-Message Information 53
5.8 The GUI Of the Tools,We Have been Trying to Construct Already 53

6.1 Histogram of percentage of each tool’s warnings 57

0.1 List of Abbreviations

BDDs	Binary decision diagrams
BMC	Bounded model checking
SAT	Boolean satisfiability problem
SMT	Satisfaction module theory
CBMC	C-Bounded model checker
ESBMC	Efficient SMT-based context-bounded model checker
DSVerifier	Digital system verifier tool
NASA	The National Aeronautics and Space Administration
CFG	Control-flow graph generator
SSA	Static single assignments
NIST	National Institute of Standards and Technology
ENISA	European Union Agency for Cybersecurity
CWE	Common weakness enumeration
FWL	Finite word-length
AFL	American fuzzy lop
CVE	Common vulnerability enumeration
LLVM	Low-level virtual machine
ANNs	Artificial neural networks

0.2 Declaration

I hereby certify that what is contained in this dissertation is the result of my effort, with the exception that reference has been made to the work of others wherever it was utilised, and affirm that neither this dissertation as a whole nor any part of it has been previously submitted for a degree or title of scientific research at any other educational or research institution.

Samih Alkhamissi
September 2019

0.3 Intellectual Property Statement

0.4 Acknowledgments

I praise my God, Allah, first and foremost for providing me with the strength to follow and to make my dream come true.

I would like to express my deepest and sincere appreciation to my supervisor, Professor Lucas Cordeiro, for his supportive words and his continuous motivation and valuable advice. His support and guidance have been key to improve my research skills.

I would also like to express my profound gratitude to my Mom and my Dad, who have prayed for me, and to my husband and my family for their motivating and encouraging words which helped me to complete my research career and offered doses of hope and patience throughout this educational journey.

CHAPTER 1

INTRODUCTION

The safety and reliability of flight-control software are essential aspects of unmanned aerial vehicles (UAVs) which are relevant to the UAV's flight control system, and have become increasingly important in military and civilian applications [11]. Companies such as NASA, Amazon and even Google have commercially experimented with this technology [12]. Any problems with the control software, affect normal the UAV system control and operation, thereby resulting in serious harm to operational missions, including recognition and intelligence transmission [13]. Strict flight control software testing and improved software safety and reliability are therefore necessary to achieve UAV quality. This research is designed and dedicated to this purpose.

Software is an increasingly critical element in aerospace systems ,and, the number of lines of software code used in missions is a criterion of growing significance [1]. The number of code lines used in the The National Aeronautics and Space Administration (NASA) missions has exponentially increased in the last few decades. Today, missions are associated with millions of code lines. In contrast, early software usage in space flight missions was traditionally associated with , against thousands of code lines [14].

There are substantial costs related to developing software and software failure [15]. A database of failures of the aerospace system demonstrates that to date, software defects have led to significant launch delays and mission failures, which have resulted in considerable costs and data loss [1]. The latest aerospace losses due to software mistakes are presented in Table 1.1. As a result,by 2012, the military department of United States of America had increased its investment in unmanned air vehicle research and production from roughly $2.3 billion to $4.2

billion in 2008 [16].

Table 1.1: Recent Losses in Aerospace due to Software Error by David et al. [1]

System	**Cost**	**Description (All involved loss of data)**
Ariane 5 (1996)	$594M	Software error shut down redundant inertial reference systems, resulting in loss of control and aerodynamic breakup.
Delta III (1998)	$336M	Software error did not account for normal roll oscillation, resulting in loss of attitude control and auto-destruction.
Titan 4B (1999)	$1.5B	Misplaced decimal point in Centaur flight software caused premature propellant depletion and deployment of payload in incorrect low orbit.
Mars Climate Orbiter (1999)	$524M	Failure to use metric units in ground software trajectory models caused steeper than expected entry trajectory and destruction while entering the Martian atmosphere.
Zenit 3SL (2000)	$367M	Software error caused premature second stage shutdown and the satellite failed to reach orbit.
Messenger (2004)	$23.9M	Software testing and other factors caused launch delays and a new launch profile requiring an additional two years of cruise time to complete the mission –with partial loss of data.

Through automated software verification and validation using formal methods, the risk of software mistakes can be reduced [17]. This study proposes two formal methods approaches for automated software verification and validation, which are model checking [18], which automatically checks if the model satisfies a specification, and static analysis,which is achieved by examining the code without running the software, [19]. The study further outlines, extensions of these approaches for integrating fuzz testing [20],which involves providing unexpected or random data as software inputs. These methods can be used in various phases of the software life-cycle to reduce risks (requirements, design and implementation).

Vulnerabilities must first be identified to preserve system integrity. A Vulnerability is defined by the National Institute of Standards and Technology (NIST) as a property of system security requirements, design, implementation, or operation, which could be accidentally triggered or intentionally exploited and result in a security failure [21]. In short, vulnerability is therefore essentially the result of one or more weaknesses in requirements, design, implementation or operation [21].

1.1 The Problem Statement

Given the many tragic and disturbing events listed in Table 1.1, which were the consequence of the notable software acceleration, there is an urgent need to ensure software quality and its integrity in order to reduce the costs of failures. Many tools have been created over the latest recent decades, but such tool still strongly require the development and integration of verification techniques in order to guarantee that the information is transmitted among them and to be sound and complete. The problem considered in this thesis can be expressed in the following question: *can an algorithmic method reason accurately about realistic UAV software to identify security vulnerabilities and simultaneously control the verification complexity?*

1.2 Outline of the Solution

This research proposes a preliminary solution through the integration of abstract interpretation as a static analysis technique with bounded model checker and fuzz testing as dynamic verification, which intends to achieve. The research suggests an interaction of data between abstract interpretation and fuzz testing, and between abstract interpretation and Bounded model checking (BMC) which represented by the use of ESBMC in order to guarantee the exchange of information.

1.3 Research Questions

This study poses the following research questions are specified to define the scope of the literature review and the project's approach. These questions are presented below:

- ***RQ1:*** What are the most common forms of security vulnerabilities in UAV's software, and which forms can be discovered using bounded model checking and abstract interpretation tools?

- ***RQ2:*** What techniques can be employed to improve the verifier's efficiency?

- ***RQ3:*** How can RQ2 technical tools be evaluated? How can UAV benchmarks be selected?

In order to address these research questions, a survey was designed according to qualitative research methodology [22]. The background study, literature review and practical experiment which serve to answer the research questions can support in building a prototype for automatic detection of software vulnerabilities in UAVs.

1.4 Aim and Objectives

The primary aim of this project is to build an automatic detection tool for software vulnerabilities in UAVs. This tool will help ensure the safety and reliability of UAVs. The results can also affirm identify the importance of integrating techniques for the detection of real-world UAV software vulnerabilities. There are five specific objectives which fall under this broad aim:

1. Achieve high code coverage of exploration for UAV software within reasonable resource limits (e.g. computation capacity and time).

2. Attempt to integrate two formal methods with fuzz testing in a novel manner, in order to achieve faster and better results.

3. Scale BMC techniques for finding security vulnerabilities in real-world UAV software.

4. Evaluate and compare the established verification tool with other existing studies.

5. Improve the quality and compliance of UAVs' embedded software with their construction requirements.

1.5 Research Methodology

In order to achieve the dissertation aim and sub-objectives listed in the previous section, the following research methodology eas adopted as outlined below:

1. Conduct literature review on recent software verification and testing techniques should be

performed for the related work.

2. Identify some common security vulnerabilities for UAVs and their main characteristics.

3. Design, build and implement a tool based on the integration of three distinct software verification and testing techniques.

4. Evaluate the proposed tool over real-world UAV software.

5. Compare the proposed tool with other tools available in the literature, which are used for the same purpose of this work.

1.6 Project Deliverables

The following results were achieved in this project:

- A project overview and plan (POP) report providing the information related to project implementation schedules, deadlines, objectives and measurements for carrying out the project.

- A prototype tool developed for an automatic detection of software vulnerabilities in UAVs software using fuzzing, BMC, and abstract interpretation.

- A dissertation report which discusses and evaluates in detail all aspects of software implemented, and analyses design improvements for future development of the UAVs' automatic detection tool for vulnerabilities..

1.7 Dissertation Structure

This research follows the methodology of design-science according to March et al. [23], which is technology-oriented and has four types of outputs and, it consists of two main activities: *build* and *evaluate*. Figure 1.1 illustrates the design-science methodology, which was employed to building the remaining research chapters.

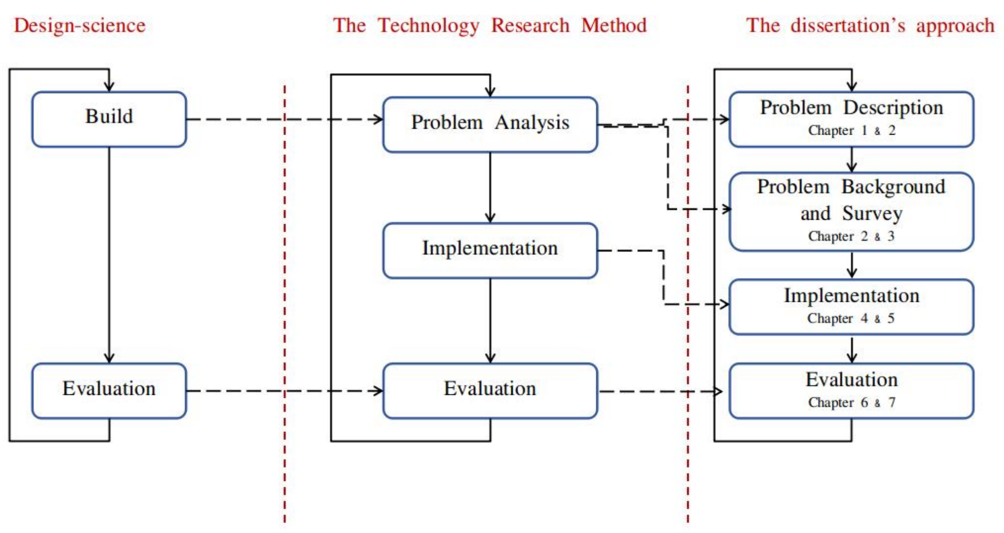


Figure 1.1: The Design-Science Approach to Research, the Technology Research Approach, and Dissertation's Approach

This research is organised into seven chapters, including the introductory chapter. Each chapter aims to address a certain aspect of the research question by separating it into tiny parts.

- ***Chapter Two*** presents the background and preliminary survey of the software verification techniques, which would attempt to comprehend the problem of UAV's safety. It highlights the concepts and also provides an overview of the well-known form of vulnerabilities.

- ***Chapter Three*** conducts a thorough analysis previously employed techniques in the related studies, which address the same subject as of this dissertation. A brief and comprehensive review of current safety tools were introduced in this section through static or dynamic or both type of analyses.

- ***Chapter Four*** describes the empirical work to related to the project's technology and architecture. It begins by describing the suggested solution in details including the comprehensive safety tool architecture.

- ***Chapter Five*** covers the details of the project implementation of the safety tool described in this dissertation.

- ***Chapter Six*** discusses the analysis and findings of the project evaluation. The chapter provides a detailed discussion in light of the analysis results and compares the project's tool related to other current safety verification tools.

- ***Finally***, the conclusion, highlights the current project achievements and limitations and offers insights for future work.

CHAPTER 2

PRELIMINARIES

Over the last two decades, several software verification and testing techniques have been proposed and significant success has been achieved in ensuring the safety and security of software [24]. These techniques typically, these techniques belong to three main categories. The first category includes techniques which are industry-standards in software engineering, such as *software testing and monitoring* [25]. The second category is *abstract interpretation and static analysis techniques*, which are regularly used remove bugs in programs with millions of lines of codes [19]. The third category involves *formal verification*, such as model checking and theorem proving [26]. This chapter explains the second and third methods and other techniques in detail.

2.1 Model Checking

The fundamental idea of model checking is defined by a model (system), which is generally defined as a finite directed graph and a specification (formal properties), which is generally defined as formula in linear-time temporal logic (LTL) [27], which are to be met by the final system. The task of the model-checking process is to automatically monitor whether or not the model meets its specification [18]. The result of the model checking procedure is either *true* when the property holds, or *false* if the property not hold; and in this case, the model generates a counterexample [24]. The idea behind the counterexample is that it can be identified, corrected, and retried as the source of the error in the model; that is, it demonstrates how the violation occurs [26]. Figure 2.1 illustrates the overall model checking structure. This chart is taken from the Embedded website [28] with some modifications on it.

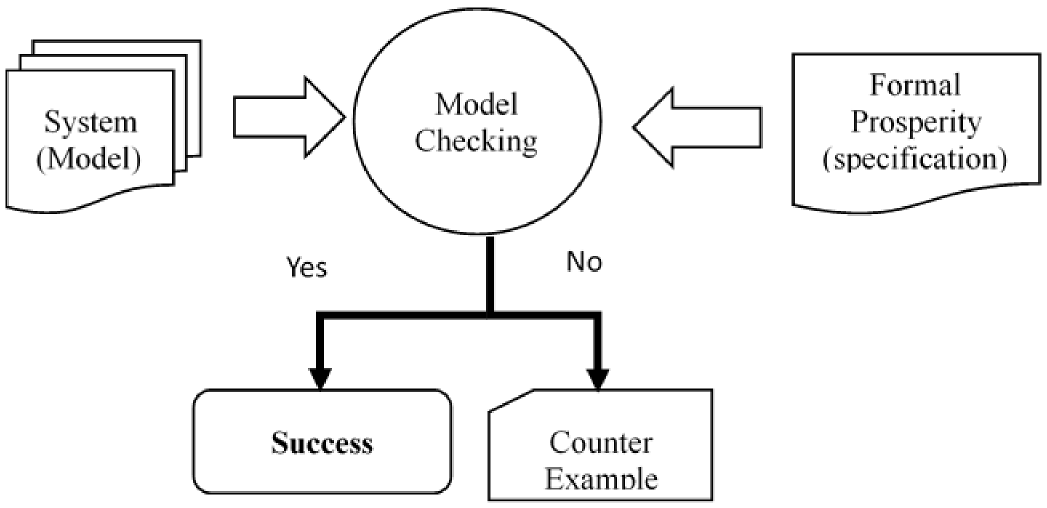


Figure 2.1: The Model-Checking Approach

In the early 1980s, Clarke and his colleagues [27] invented the term ’model checking’. The first model checking algorithms were used to check the system’s available state explicitly in order to check the accuracy of a specific specification. As the number of states can grow exponentially in terms of the number of variables, early implementation could only deal with small designs and did not include examples of complexity.

2.2 Bounded Model Checking of Software

Instead of attempting to transform the software into a state machine to be assessed, one can instead apply BMC as the systems become more complex, as outlined by Biere et al. [18]. BMC is a model checking method that checks the model up to a given path in the path length. Its algorithms traverses a finite state machine for a fixed number of steps ,k, and checks whether a violation occurs within this time step. As such, BMC is looking for a counterexample in executions whose length is restricted by fixed number of steps, k . If no bug is found, k is increased until either a bug is identified, the problem becomes uncompromising, or some pre-known upper bound is reached [29]. Figure2.2 illustrates this idea.

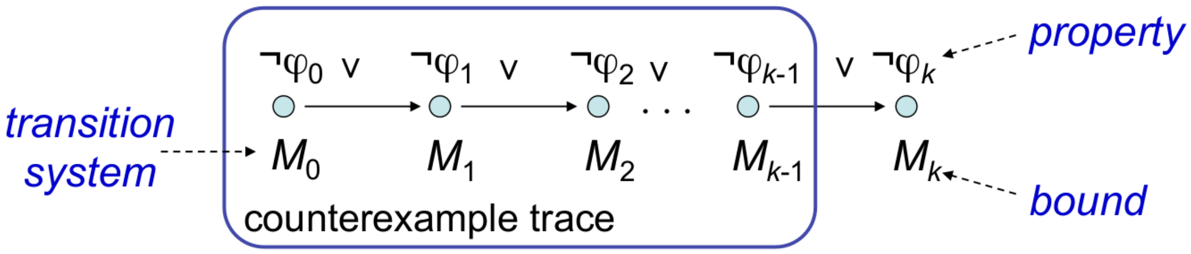


Figure 2.2: Bounded Model Checking Approach by [3]

Although BMC attempts to resolve the same problem as traditional BDD-based symbolic model checks, it has two specific features. Firstly, the user must first provide a bounding basis for the number of cycles to be investigated ,k, which means the method is incomplete when the boundary is not sufficiently high. Secondly, rather than Binary decision diagrams (BDDs), it uses Boolean satisfiability problem (SAT) techniques, where SAT solvers can address hundreds of thousands of variables [29]. The fundamental concept is to find a counterexample of a certain length and to create the propositional formula, which will satisfy such a counterexample. The efficiency of this approach is based on the observation that only a fragment of the state space is sufficient to identify an error if the system is defective [30].

2.3 ESBMC

ESBMC [31] [4] is the first SMT-based, context-based and open-source model checker that allows the verification of single- and multi-threaded ANSI-C code to check both data structure manipulation and user-defined properties (as stated in the program) automatically. ESBMC is primarily aimed at helping software developers of software by discovering subtle bugs in their code [4]. Although, no unique annotations in the software are required to locate these bug, but it allows developers to add their own assertions and also to check for violations. ESBMC has achieved significant success after being used in a number of telecommunications, control systems and medical devices applications [32].

ESBMC [4] uses the clang [1] [33] , a state-of-the-art compiler suite for C/C++/ObjectiveC /ObjectiveC++, as front-line in order to prevent the need to maintain a separate front-end to instead concentrate on the main objective of software verification. It uses the floating-point, either using the SMT theory of floating-points or using bitvectors, as back-end. Furthermore, ESBMC uses k-induction to demonstrate the absence of property infringements.

[1] http://sourceforge.net/projects/ctags

2.3.1 Features

ESBMC's basic architecture is illustrated in Figure 2.3. The tool takes the C source code as input. The remaining steps are as follows:

1. ***Front-End***: As previously stated, ESBMC relies on the clang's API as its front end, which is used to convert clang's AST into ESBMC's own AST. This component makes ESBMC dramatically different from LLBMC, which relies on LLVM bytecode. The disadvantage of ESMBC is that polymorphism and some other minor features are not supported because C++ elements of the clang AST are still not fully integrated with ESMBC.

2. ***Control-flow Graph Generator***: After generating AST, clang converted it into an equivalent GOTO program, which eliminates all *for*, *while*, *do-while* and *switch* statements, and adds checks for zero division and out-of-bound access.

3. ***Symbolic Execution Engine***: The symbolic execution engine unrolls the loops of the source code k times and generates the Static single assignments (SSA) program form. The SMT solver checks all safety properties derived during the symbolic execution. ESBMC simplifies the program aggressively, with constant folding and various arithmetic simplifications, to create small SSA sets.

4. ***SMT Back-End***: This Back-End supports five solvers: Boolector [34] (default), Z3 [2] [35], MathSAT[36], CVC4 [37] and Yices [38]. ESBMC uses the back-end to encode a SSA form of the program in a quantifier-free format and to check satisfiability of $C \lor \neg P$, where C is the set of constraints and P is the set of properties. The program includes a bug if the formula is satisfiable: ESBMC will produce a counterexample with the set of tasks, which led to the property violation.

5. ***Python API***: ESBMC now provides a Python API, which decreases prototyping complexity and makes the internal tool more available for the general public. However, there are also drawbacks in the Python API, which is slower than C++, and developers can perform illegal operation, thereby causing the tool to crash.

[2]https://z3.codeplex.com

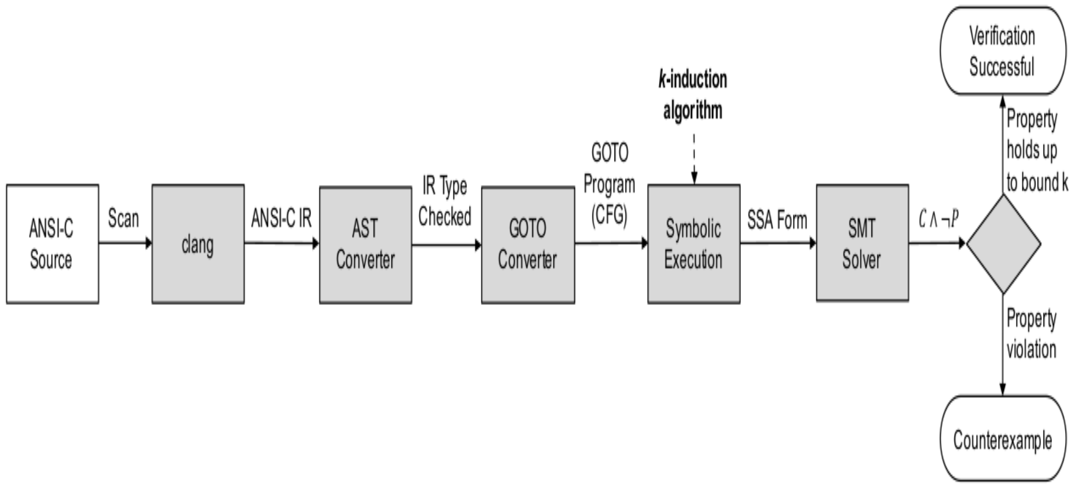


Figure 2.3: ESBMC Architecture by [4]

2.4 Abstract Interpretation

Patrick Cousot and Radhia Cousot proposed abstract interpretation in 1978 [39]. The main idea of the abstract interpretation is to define an approximation of the semantics of a program. Formal proof can be made of the program at this different level of abstraction, where irrelevant details are removed to reduce the complexity of the testing process [40]. Cousot [41] defined abstract interpretation as an abstraction and constructive approximation theory of mathematical structures, which are used in formal program language descriptions and inferences or verification of undecidable program properties. Static analysis by an abstract interpretation is a technique that attempts to demonstrate a lack of run-time error through the analysis of a program's source code. The types of errors typically checked include buffer overflows, invalid pointer accesses, array boundary checks, and arithmetic underflow and overflow [42].

In systematic method construction and efficient algorithms, abstract interpretation can be introduced for the approximation of undecided or very complicated issues of computer science such as semantics, proofs, static analysis, inspection, safety and security of software or file systems [43]. Its objective is to solve such problems by achieving complete program coverage, by analysing the behaviour for all possible inputs. The value of this method is that there are only approximately achieved outcomes. That is, a vulnerability-finding analysis with false-positive results can be generated and all vulnerabilities are discovered. The abstract analysis findings are regarded to be as sound as their mathematical proof [44]. Abstract interpretation is based on a function flowchart representation, which is a Control-flow graph generator (CFG). Most existing numerical abstract domains such as intervals [45], octagons [46], polyhedra [47], can only express nonlinear sets. In programs where control-flows are combined, an abstract domain often uses a joint operation to abstract the interruption (union) of nonlinear constraints from incoming edges into a combination of new nonlinear constraints [48]. This pattern is clarified by an example illustrated in Figure 2.4. A control flow graph is used to minimise the absence of dynamic execution information. This graph demonstrates the flow of system functions and can be used to analyse constraints for different values within the system.

AI Example. CPP

```
void function()
{
        int a = 1;
        int b = 2;
        if (b == 2)
        {        ++b;          }
        int c = 3;
        int d = 4;
        while (a < 5)
        {        ++a;   }
        int e = 5;
        int f = 6;
}
```

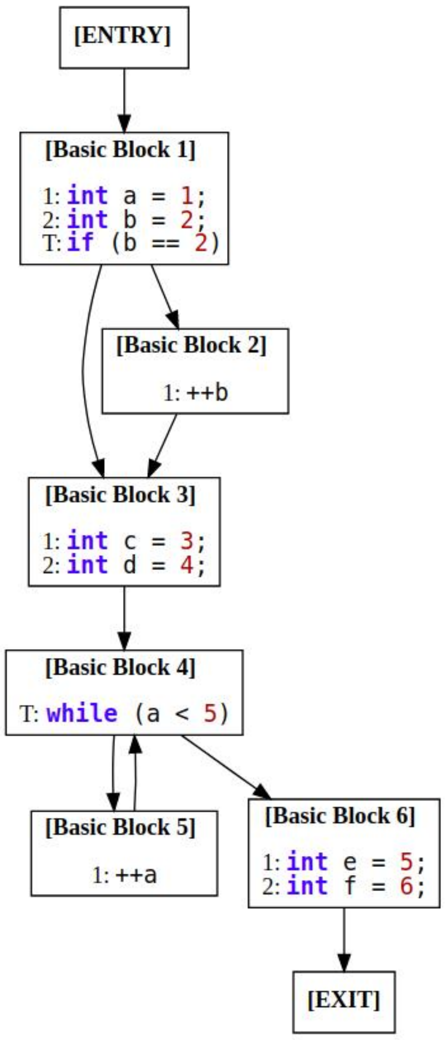


Figure 2.4: An Example of a C++ Code and its CFG [5]

2.5 Fuzz Testing

Fuzzy is a black-box testing technique. Fuzzy testing discovers the software issues, which are often triggered by inputs that would have been unexpected to be constructed by a developer or data structures through external interfaces [49]. Takanen et al. [49] claimed that fuzzing does not attempt to verify a system but rather seeks to discover faults to detect as many vulnerabilities as possible. By generating test cases and monitoring whether the program is in a crash, fuzzing generates a piece of evidence for each crashing [50].

The first formal mention of fuzzing was made by the Research Project of the University of Wisconsin-Madison, and fuzzing has since been adopted to define a whole software testing methodology [20]. In 1988, Miller et al. [51] first suggested fuzzing in 1988, and since then it has become a powerful, fast and practical technique for locating vulnerabilities in software [52]. Fuzz was has generally been implemented effectively for web applications [53] and compilers [54].

However, the main concept of the fuzz test is to generate semi-valid test data [51] . Sutton et al. [20] splits fuzzing based on the input injection into two categories; mutating current, valid data or producing data with specific regulations. Fuzzers which alter current test cases in order to produce fresh ones are referred to as mutation-based fuzzers, while fuzzers which themselves generate test cases are deemed generation-based fuzzers. [51]. Table 2.1 demonstrates a comparison of fuzzing data generate.

Table 2.1: Comparison of Fuzzer Based on Generation and Mutations by [2]

	Priori knowledge	**Coverage**	**Ability to pass validation**
Generation-based	needed, hard to acquire	high	strong
Mutation-based	not needed	low, affected by initial inputs	weak

Jääskelä [6] categorised fuzzy testing as *a black-box, a white-box and a grey-box* , depending on how much information the target program needs in run-time. The first method is called *black-box* fuzzing, as there is no need for an understanding of target behaviour.In contrast, *white-box* fuzzing assumes full knowledge of application code and behaviours such as modelling the target. Meanwhile, *grey-box* fuzzing, for which the target software is assumed to be partial knowledge, is located between the two previous techniques to make use of both [55].

The slight distinction between these types is further described in Figure 2.5.

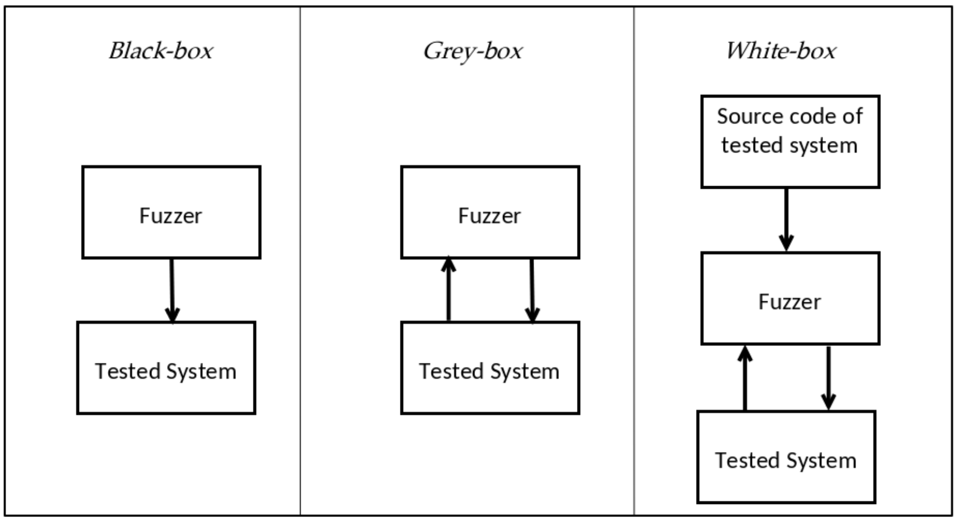


Figure 2.5: The Multiple Sorts of Fuzzing by [6]

2.6 Vulnerabilities Overview

Common weakness enumeration (CWE) [56] is a software weakness and vulnerability category system. It is supported by a community project to understand software defects and create automated tools for identifying, correcting and preventing such defects. The project was supported with assistance from the US-CERT and the National Cyber-security Division of the US by the National Cybersecurity FFRDC, which owns The MITRE Corporation. On June 2019, version 3.3 of the CWE Standard was published [57].

Vulnerabilities are defined as the properties of system security requirements, design, implementation, or operation, which could be accidentally triggered or intentionally exploited and thereby result in a security failure [21]. In most cases, vulnerabilities fall into two categories: bugs at the implementation level and flaws at the design level [58]. Bugs can be classified as shallow bugs and hidden bugs. The bugs that cause the target program to crash during initial execution are considered shallow (e.g., a possible division-by-zero operation with no precedent branch condition). Conversely, bugs that occur very deeply in the logic of the program and are difficult to activate are considered hidden bugs, such as bugs that are found in complicated conditional branches [59].

In 2016, the world economy spent 1.1 trillion dollars due to software defects. Software faults were identified in 363 enterprises, impacted 4.4 billion clients, and created a loss of moment over 315 years. Most of these events could have been avoided, however, without adequate testing, the software was merely moved to manufacturing [60].

Although significant scholarly and applied research has tackled the safety concerns of software in recent years, hundreds of fresh vulnerabilities are detected, released or utilised monthly [61]. From 2010 to 2015, around 80,000 vulnerabilities were recently registered in Common vulnerability enumeration (CVE), and there have been further increases in vulnerabilities since [62]. It is clear that, attack types, impacts and vulnerabilities are changing, rapid as illustrated in Figure 2.6. The increasing number, type and effect of vulnerabilities over three years can clearly be identified, where circle size estimates the relative effect of incidents in terms of business costs.

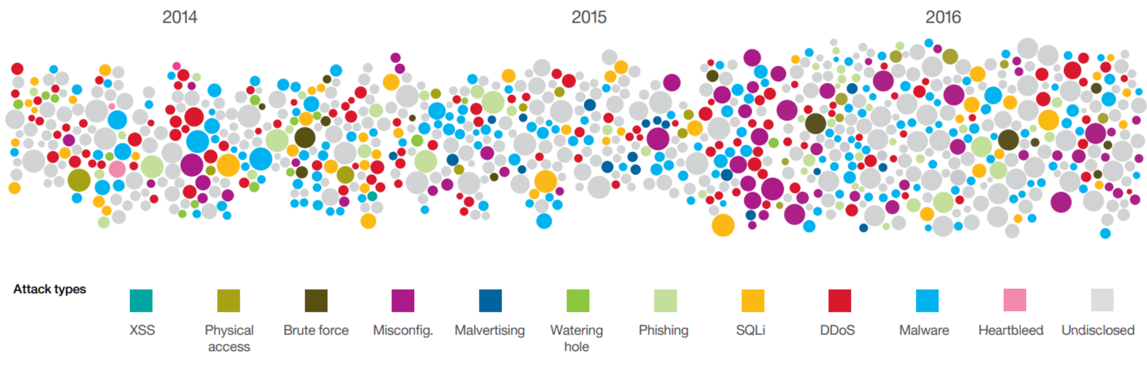


Figure 2.6: Sampling of Security Incidents by Attack Type, Time and Impact, 2014-2016 by [7]

Because of the difficulty of defining all types of vulnerabilities, this dissertation does not describe all feasible vulnerabilities. Instead, it focuses on certain hazardous security vulnerabilities, which are extremely exploitable and have a strong effect.

2.6.1 Buffer Overflow

Over the past ten years, buffer overflows have been the most prevalent type of security vulnerability [63]. The most frequent reason for buffer overflow assaults is because apps fail to handle memory allocations and validate customer or other procedures input [21]. These types of vulnerabilities Often allow remote attackers to execute arbitrary code on the target server or to crash the server's software to attack the service denial (DoS). Buffer overflows represent approximately $1/3$ of the serious remotely exploitable vulnerabilities in the NIST ICAT database [64]. The European Union Agency for Cybersecurity (ENISA) [65] defines buffer overflow as a vulnerability bug in a computer which occurs where a program or process attempts to write or read more data from the buffer than a buffer can hold. As a buffer is a sequential part of the memory that is temporarily used to store data, extra data in memory rows next to the target buffer can be overridden unless the program contains sufficient limits to control flag or drop data if an excess amount is being sent to a memory buffer[66].

In a security context, ENISA [65] claims that a buffer overflow can allow the attacker to access different parts of the internal memory and eventually control the execution of the program, creating a confidentiality, integrity and accessibility risk, which represent a paradigm for the assistance of information security policies inside an organisation also known as the CIA triad. Confidentiality is a set of regulations limiting access to data, integrity ensures that the data is trustable and accurate, and accessibility ensures secure access by an authorised entity to the data [25]. Figure 2.7 provides an example of buffer overflow, where a string with a length larger than 30 characters leads to the memory address corruption, including the return address of the last stack function. The corrupted return value will lead to a segmentation failure when the current function returns [67].

Buffer Overflow.C

```c
int main ( )
 {
   char name [ 3 0 ] , ch ;
   int i =0;
   printf (" Enter name : " ) ;
   while ( ch != '\n')
             { // terminates if user hit enter
                ch=getchar ( ) ;
                name [ i ]= ch ; // crash !
                i ++;
             }
  name [ i ]= '\0'; // inserting null character at end
  printf ("Name : %s " , name ) ;
  return 0 ;
 }
```

Figure 2.7: A Buffer Overflow Example Code

According to the IBM X-Force Threat report in 2017 [7] thirty-two percent of the attacks comprised data structure processing, which attempted to achieve unauthorised access by manipulating system data structures. Many vulnerabilities are caused by ambiguity and assumption in design and prescribed manipulation such as, buffer overflow vulnerabilities, and thus the exploitability of these data structures. Figure 2.8 illustrates top attack types for monitored security from January 1st 2016 through December 31st 2016.

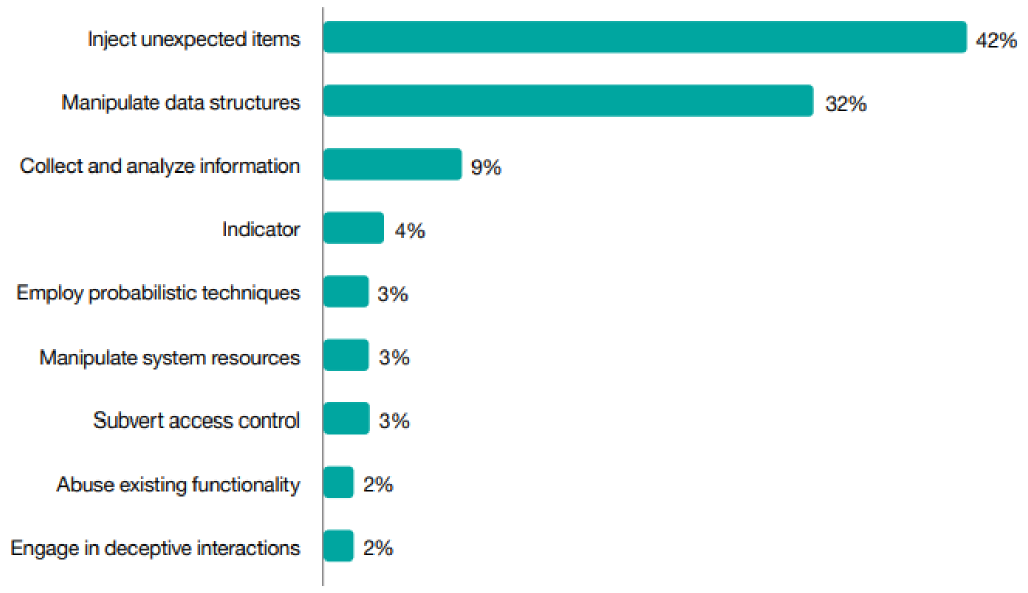


Figure 2.8: Top Attack Types for Monitored Security by [7]

2.6.2 Integer Overflow

Integers[3] are data types that are allocated as numerical values. An integer is allocated memory statically at the declaration point; the amount of memory allocated to the integer is dependent on the host hardware and operating system architecture. Integer overflow or flow indicates that the dynamically assigned memory request is far too large or too small for the memory address required by the software. [68]. The NIST [56] describes the integer overflow vulnerability as it exists when calculations try to increase a higher integer value than the integer used to store in the corresponding representation. The integer value can be converted into a negative or very insignificant amount when this fault occurs. This vulnerability is thus crucial for safety if the calculation results are used to handle loop controls, as well as to determine the size or offset of behaviours (e.g. concatenation, copying, allocation of memory).

The renowned space shuttle failure, Arian 5, was due to the fact that the software in the inertial reference system was shut down in response to an untreated number exception which was the 64-bit floating-point conversion to a 16-bit signed integer value (integer overflow) [68]. This event cost nearly 500 million dollars [1]. Figure 2.9 provides an instance code for this vulnerability form.

[3]https://www.iso.org/standard/18939.html

Integer Overflow.C

```
int main(void)
        {
            int l;
            short s;
            l = 0xdeadbeef;
            s = l;
            printf("l = 0x%x (%d bits)\n", l, sizeof(l) * 8);
            printf("s = 0x%x (%d bits)\n", s, sizeof(s) * 8);
       }
```

Figure 2.9: An Example Code for Integer Overflow by [8]

2.6.3 Division By Zero

Dividing a number by zero is a mathematical error which leads to a run-time error by an unspecified outcome [56]. When a code is written without exception handling, the division-by-zero output will be demonstratedshown to be 'plus infinity' or 'minus infinity', which can not be further processed according to the IEEE 754 standard [69]. However, the variable does not necessarily have to mean zero: the variable may be called arithmetical underflow as a consequence of value rounding. In this case, the results are assumed to be equal to 0. A division-by-zero error in the remote data base manager [4] on the USS Yorktown brought all of the devices down on the network and caused the propulsion systems of the ship to fail. A crewman had entered in a database a blank field in a database. The blanks were considered zero and caused a division-by-zero exception, which can not be handled by the database programme. The exception aborted the Microsoft Windows NT 4.0 operating system, which crashed and brought down all of the LAN controllers and remote terminals of the ship [68]. Figure 2.10 provides this exception with an instance.

[4]https://www.patriotspoint.org/explore/uss-yorktown/

Division By Zero.CPP

```
const int DivideByZero = 10;
double divide(double x, double y)
{
    if ( 0 == y )
    {
         throw DivideByZero;
    }
    return x/y;
}
```

Figure 2.10: A Divide By Zero Code by [9]

2.6.4 NULL Pointer Dereference

CWE [56] describes A NULL pointer dereference as it takes place where the application dereferences a pointer that it expects to be valid but is a NULL that typically causes an application to crash or exit. CWE claims that the NULL pointer dereferences generally lead to a system error unless exceptional processing (on certain systems) is accessible and implemented. It can still be challenging to restore the software to a safe state of service even when exception processing is utilised. Figure 2.11 provides an example for null pointer; when this code is compiled and executed, it produces Zero address, which means that a first pointer in the memory, as the memory is reserved this location by the operating system, the software can not access memory at it.

NULL Pointer Dereference.C

```
int main ()
 {
   int  *ptr = NULL;
   cout << "The value of ptr is " << ptr ;
   return 0;
 }
```

Figure 2.11: Possible NULL Pointer Dereference Vulnerability by [10]

CHAPTER 3

SYSTEMATIC LITERATURE REVIEW

This systematic literature review was conducted according to the guidelines proposed by Booth et al. [22] and Kitchenham and Charters [70], who are among the most widely recognised authors in the science and software engineering domains, respectively. The review starts by identifying the need for the literature review and proposing sub-objectives, which are relevant to the research aims. In addition, state-of-the-art tools are defined throughout this section.

3.1 Literature Review Objectives

The aim of this literature review is to (1) summarise existing studies on software testing and verification tools to gain insight into the present scenario concerning software quality in UAVs, (2) examine the most recent methods in this field and (3) identify research gaps and inconsistencies in previous studies.

3.2 Search Strategy

In order to produce a balanced, systematic literature review that is applicable to specific information relevant to the research questions, it is necessary to gather all appropriate studies [70] across searching through the search strings, which are a composite of search terms, as described in Table 3.1. As this study concentrates on the investigation of UAVs software, the term UAV is considered to be the cornerstone of the research process. In order to strike a balance between broad and specific searches, the Boolean operators (AND, OR) were used by combining search terms to broaden or narrow search results. Furthermore, the Wildcard operator which is represented by a star sign ‘*’ has been used for distinct spellings such as ‘Vulnerability’ and ‘fuzz*’ terms. Table 3.1 below lists various terms that were used for the search strings.

Table 3.1: Keywords and Search Terms

Search terms	
A1. Software verification tools	B1. ESBMC
A1. Formal methods	B1. k-induction verifier
A3. Bounded model checking	B3. Vulnerabilit*
A4. Abstract interpretation	B4. swarm testing
A5. Fuzz*	
A6. UAV	

Key search strings

- (A1 AND A6 OR B1 OR B2 OR B3 OR B4).
- (A2 AND A6 OR B1 OR B2 OR B3 OR B4).
- (A3 OR A4 OR A5 AND A6 OR B1 OR B2 OR B3 OR B4).
- (B1 OR B2 OR B3 AND A6 OR B4).

3.2.1 Data Resources

This project selected digital libraries, which contain academic literature relating to software engineering, some of whichwere accessed through the portal of The University of Manchester. A list of these libraries is provided below:

- IEEE Xplore http://ieeexplore.ieee.org/Xplore
- SpringerLink http://link.springer.com
- ACM DL http://dl.acm.org

- Google Scholar https://scholar.google.co.uk

3.3 Study Selection

Following the collection of the potentially relevant primary studies, the relevance of these should be assessed based on a set of inclusion criteria, the following criteria can be defined:

- Criteria 1: The research or resource should be in the English language and related to the computer science field.
- Criteria 2: Software verification must be explicitly considered in the underlying research.
- Criteria 3: One or more of the following verification methods must be addressed in the underlying research.
- Criteria 4: The research project should directly, indirectly, and highly relevantly relate to the study or resource.
- Criteria 5: Relevant studies need to be carried out recently and after 2000.

3.4 Study Quality Assessment

A quality assessment checklist was developed to evaluate the quality of each study to measure each individual survey based on its comprehensiveness for the subject. The Quality Assessment Checklist for each publication is described in Table 3.2.

Table 3.2: Checklist for Resource Quality Assessment

Checklist Question		
Does a resource cover any software verification technique?	○ **YES**	○ **NO**
Are any software verification issue addressed by a resource?	○ **YES**	○ **NO**
Does the resource give some insight or link to the research study?	○ **YES**	○ **NO**
Is there any subtopic linked to the study project covered by the source selected indirectly?	○ **YES**	○ **NO**

3.5 Search Results

Following a search of the above-mentioned databases, 21797 results were retrieved (see Table 3.3), which were journal articles and conference papers and 18 papers were retained after applying the criteria specified in the Study Selection and Study Quality Assessment sections 3.4 on the title and the abstract of the scientific paper,includingwith three pre-selected studies. Each selected study satisfied all the inclusion criteria by assessing each publication throughout the checklist of quality evaluation to prevent undesirable confusion.

Table 3.3: Digital Library Search Results Using Search Strings

Digital libraries	**Primary Studies**
IEEE Xplore	69
SpringerLink	212
ACM DL	416
Google Scholar	21,100

3.6 Review of Software Vulnerability Detection's Studies

Software vulnerability research can be classified into two different categories: analysis of software vulnerability and finding of software vulnerability [71]. Moreover, software vulnerability detection techniques can first be divided into two approaches: static analysis and dynamic analysis. Static software analysis is a debugging method which is performed in the source code or binary compilation without executing them. Dynamic analysis includes a run-time examination of the software [72]. Software techniques for vulnerability detection have grown from initially pure manual discovery through to computer-aided discovery and continue moving in a fully automated direction [71]. The subsequent sections examine existing studies related to these techniques.

3.6.1 Bounded Model Checking Verification

Model checking is an efficient method for discovering software bugs. Compared to testing, the model checking can demonstrate that no bugs exist while testing can only discover the bugs not prove their absence. Model checking provides better coverage for this comparison, but is more computationally expensive [73].

Humphrey et al.[17] argued that formal techniques such as model checking might be helpful for the UAV mission planning domain. If mission objectives and limitations – for example, tasks which have to be coordinated, areas that must be monitored in a given order and areas which must be avoided – can be formalised into specifications, and UAV actions as finite-state models, then model checking can be used as a tool to assist operators to track mission plans and error reasons with counterexamples. NASA employed model checking in their project, by developing JPF as the second generation of a Java model checker [74] after the tragic events of its missions when systems did not perform as per specifications. The UNO tool employs the model checking strategy, which Holzmann et al.[75] proposed as a solution for the most prevalent defects types in C-programs using uninitialised variables, null-pointer dereferencing and buffer overflow indexing. The UNO checking capabilities can be enhanced by defining application-dependent properties, which are written as ANSI-C processes by the user.

However, Corbett et al. [76] argued that to utilise a verification tool in the real program, the developer must extract an abstract mathematical model from the high-level properties of the program and specify this model in the input language of the verification tool. It is an error-prone and time-consuming process. The state explosion problem is further barrier in the transfer of finite-state variation technology: the exponential increase in finite-state size as the number of systems components increases.

Rocha et al.[77] proposed DepthK, which is a software verification tool using BMC and *k*-induction based on invariants of program generated automatically by using polyhedral restrictions. As their primary verification engine, DepthK utilises Efficient SMT-based context-bounded model checker (ESBMC), a context-bounded symbolic model checker to verify single-

and multi-threaded C programs. In particular, ESBMC is used either to detect property violations up to a specified bound k or to indicate correctness using the k-induction scheme. The experience findings indicate that 1,091 confirmed TP outcomes have been achieved and 1,056 FP confirmations have occurred, with a further 467 unconfirmed outcomes, as well as 20 TN outcomes and 32 FN outcomes, mainly due to the constraints in the ESBMC memory model. In addition, Mikhail et al. [78] developed a new generator for interval invariant, which preprocesses the program, infers invariants based on intervals and introduces them as assumptions in the program. Their findings indicate that ESBMC v6.0 can prove up to 7 percent more programs by k-induction when invariant generation is enabled. They claimed that, the k-induction algorithm is an effective verification technique implemented in different software model checkers to prove partial correctness across a wide range of programs and properties.

3.6.2 Abstract Interpretation Verification

Abstract interpretation allows researchers to relate standard static analysis techniques to dynamic tests through verification of certain dynamic properties of the source code without running the program [44]. The difference between static analysis and dynamic analysis techniques is that the static technique analyses the software and its source code without the need to run it. The depth of the analysis creates control flow graphs and analyses large units in order to provide approximations of how the system operates during actual implementation [79]. Cousot [41] claimed that abstract interpretation has a wide range of applications from the theory to practice. Static analysis based on abstract interpretation is automatically, accurately and commercially supported to prove abstract run-time errors, sound, scalable and industrial-size software. Bouissou et al. [43] argued that in recent years, static analysis through abstract interpretation has been very successful in automatically verifying complicated properties of safety-critical embedded systems in real time.

VTSE [80] collects two inputs, which are the source code of the program and a user assertion. Its outcome is a report on the satisfaction of the user's assertion. Following the collection of the source code, VTSE symbolises the symbolic execution in a sequence of steps, including creation of the abstract syntax tree, construction of control flow graph, unwinding loops, then indexing code variables to form a metaSMT formula, which will be transformed into a first-order logical formula. VTSE combines the abstraction structure of the source code with the user assertion and reaches the final form in a solver for SMT. Two SMT solvers candidates are

Z3 [1] and raSat [2]. VTSE has a generally beneficial effect, as well asincluding on the number of problems that have been solved compared to CBMC, when checking programs with big code lines.

Static analysis is the only technique that eliminates buffer overflows and their impacts; it can be used in open-source software for large numbers of C open-source projects. However, the problem of static analysis techniques is that either false negatives or false positives tend to be generated because the application is not actually running. One false alarm for every 30 to 60 lines of code in a block of 20-40k lines of code is the upper limit of the false alarm rate obtained from the amount and type of alarms produced by Polyspace on NASA software [64]. BOON [81] has developed as standard static analysis techniques for detecting buffer overflow in C source code in order to identify and fix vulnerabilities actively, prior to their are exploitation. This sort of vulnerability is prevalent in C due to unsafe and string operations. Wagner et al. resolved all standard library operations for string manipulation by models, with the quantity of memory assigned and the number of bytes used for the analysis of the integer array. Its formulated as an integer constraint for each declaration in the system and a warning is provided after the resolution of the restriction system if a declaration violates the constraints.

3.6.3 Verification by Fuzz Testing

Today, the use of fuzzing is one of the most efficient ways of recognising software vulnerabilities. In 2010, VUPEN Security is a leading vulnerability research company providing advanced security vulnerability analysis and exploitation, which allows companies to protect against cyber-attacks, which are any type of deliberate exploitation that targets information systems, infrastructures, networks and personal computer devices. VUPEN detected 147 critical code execution vulnerabilities in major software programs such as Microsoft Office and Adobe Acrobat Reader. Many of these vulnerabilities have been discovered by fuzzing [55]. For example,Microsoft discovered and fixed over $1,800$ bugs using a distributed fuzzing framework, with more than 800 million iterations in over 400 formats [82].

In 2008, Guang−Hong et al. [50] presented ”GAFuzzing” as a vulnerability analysis approach in executable program which brings together static analysis, dynamic analysis and genetic algorithm to detect buffer overflow vulnerability in C code. GAFuzz disassembles code then, scans for vulnerabilities and calculates the path to vulnerability, which feeds inputs to the

[1]https://github.com/Z3Prover/z3

[2]https://github.com/tungvx/raSAT

program after static analysis. The findings of their experiments indicate that for vulnerability analysis and penetration testing, which is a practical test to detect security vulnerabilities exploited by an attacker, GAFuzzing is better than random fuzzing, which sends random data for the target program to detect these vulnerabilities [50].

Shallow and hidden bugs can not be recognised as standard; therefore, the frequently used evaluation criterion of the fuzzer is the code coverage (i.e., the number and exploit ability of bugs found) [59]. Indeed, the primary disadvantages of fuzz tests are its weak coverage, which entails many bugs, and the quality of tests. Identically, the other disadvantage that fuzzing attempted to solve is the blindness during a test case generation phase, which can lead to a low level of code coverage [59]. Improving fuzzing through effective methods such as data tainting and coverage analysis may enhance its effectiveness and make it smarter [61]. Bekrar et al.[61] proposed a theoretical approach by integrating fuzzy and data tainting into the assembly level, in order to improve coverage effectiveness, where data tainting improves fuzzing to identify the most promising test sequences that can cause potential vulnerabilities and narrow the test space using vulnerability information such as path execution. Smart fuzzing, which means generating the format-appropriate input values by target software analysis and error generation, has the ability to know where errors can occur via software analyses. However, there is a disadvantage in that expertise is needed for analysis of the target software and that generating a template for software input takes a long time [62].

3.6.4 Hybrid Verification

The combination of verification methods is an exciting strategy to overcome the disadvantages of the aforementioned dynamic and static verification methods. The primary focus of the hybrid verification strategy for embedded software is on combining model verification and theorem theorems such as SMT and predicate abstraction methods [83]. Hendrik et al. [42] adapted the original Orion idea [84] by replacing the lightweight data-flow analysis (DFA), which is a type of static analysis technique that typically operates over a CFG, by the abstract interpretation tool Polyspace[3]. Instead of the solvers CVC and Simplify, which are used in Orion, the researchers employed C-Bounded model checker (CBMC) to reduce the number of false positives. It is attractive and cost-effective to combine different verification methods. More than 20% of Polyspace's warnings were automatically discovered by their experience.

for a value analysis of floating-point programs, Ponsini et al. [85] proposed a hybrid ap-

[3]https://www.mathworks.com/products/polyspace.html

proach. In a single static and automatic analysis, use combined abstract interpretation and constraint programming techniques for avoiding a combination explosion in the number of paths to be explored. Ponsini et al claimed that, the hybrid approach is slower, but more accurate. Furthermore, a hybrid method can calculate approximations of program variable values, for as neither the abstract interpretation nor the constrained programming technique can indeed calculate it alone.

In order to analyse the sanitiszation process, Saner [86], based on the current tools, created static Pixyand and dynamic analysis techniques to recognise defective sanitiszation processes, which can be disrupted by an attacker on the web application. The concept behind its integration is to produce a more complete and sound tool by checking whether the identified sanitiszation is accurate and complete. Then tool was applied to a number of applications in the real-world. Its findings indicate that Saner was able to detect several new vulnerabilities stemming from incorrect sanitiszation processes.

The hybrid fuzzing technique combines the advantage of fuzzing to produce random input values and concolic execution to check the program execution path. The hybrid fuzz solves the incompleteness of fuzzer and the concolic execution path explosion problem [62]. Driller [87] is a novel hybrid fuzzing tool, which combines a genetic input mutating fuzzer with a selective concolic execution engine to recognise profound bugs in binaries. Stephens et al. [87] claim that the combination of these two methods enables Driller to operate in a scalable manner and to bypass the input test cases requirements.

3.6.5 Cutting-Edge Software Verification Tools

With respect to research carried out on recent verification tools, Digital system verifier tool (DSVerifier) [4] which achieved the success rate exceeded by American fuzzy lop (AFL)[5] as described by Chaves et al. [88]. DSVerifier is an application of incremental bounded model checking and a *k*-induction tool which is concerned with the verification of digital system implementations; it was the first tool to investigate Finite word-length (FWL) effects in UAVs. By searching for implementation errors related to FWL effects in UAV digital controllers, which occur because these properties usually take into account the complex dynamics of the systems and require tools for the verification of hardware that are specialised in implementation. DSVerifier achieved a significant result for investigating overflow and LCO in 10 different dig-

[4]https://ssvlab.github.io/dsverifier/
[5]http://lcamtuf.coredump.cx/afl/

ital controllers, through 84 different implementations.

AFL [89] is a security-based fuzzer that uses a novel form of compile-time device and genetic algorithms for the automatic discovery of test cases which are triggering fresh internal conditions in the binary. This technique improves the functionality of the fuzzed code substantially. AFL [90] performs file format mutation fuzzing, so that valid input file instances are first seeded. It then measures the code coverage of each test case and uses genetic algorithms to develop a group of test cases covering the maximum code size. This approach is quite successful in real-life fuzzing campaigns, with AFL finding many serious vulnerabilities in many popular applications.

Astrée developed by the École Normale Supérieure and the CNRS [91] as stands for real-time embedded software static analyser. It is a program analyser which is static, entirely automatic, semantic-based, sound and has proven to be effective in reality. It is a static program analyser designed to demonstrate that run-time errors are not present in programs published in C programming language. Astrée claims any zero-division, out-of-bound array indexing, wrong handling and dereferencing of the points, integer and floating-point arithmetic overflow. It was successfully used for large-scale integrated control-command safety software, which was automatically produced from synchronous requirements and produced an accuracy proof without any false alarm for complicated software within a few hours of calculation. It is very quick and extremely accurate for floating-point computations [92].

Table 3.4 summarises all verification tools under study in terms of language support, the target and key techniques.

Table 3.4: Summary of Cutting-Edge Software Verification Tools

V.TOOL	**LANG.**	**VUL. TYPE**	**TARGET**	**TECHNIQUES**
DSVerifier	C/ C++	The first tool to investigate finite word-length	Digital systems	Bounded model checking Based on SMT solver
AFL	C		Desktop application	Gray-box fuzzy testing And genetic algorithms
Astrée	C	Divisions by zero, Buffer overflows, Dereferences of null or dangling pointers, data races, deadlocks	Defense/aerospace, industrial control, electronic, and automotive industries	A static program analyser based on abstract interpretation

Chapter 6 presents a further comparison of these tools.

3.6.6 Summary and Conclusion

Table 3.5 summarises all the techniques under consideration being studied in term of the advantages and disadvantages of each technique.

Table 3.5: Comparison of State-of the-Art Techniques

Techniques	**Pros**	**Cons**
Bounded Model Checking	• It finds counterexamples very rapidly because of the initial thoroughness of the SAT search procedure. • It discovers minimal length counterexamples. This mechanism enables the user to better comprehend a counterexample. • It uses less space than approaches based on BDD.	• All possible execution paths must be encoded into one SMT formula, resulting in a number of limitations to be checked. • It is suffering from the explosion issue of state space.
Abstract Interpretation	• An abstract tuning is not required because an approximation of its semantics is given to the program model.	• False alarms are generally created by overapproximating potential executions of the program.
Fuzzing Testing	• The test has a high degree of automation. • Fast and simple idea. • It does not produce a false positive.	• Fuzzy testing requires a massive input space. • The target software requires time-consuming analysed that requires a long time. • There is weak coverage of the testing.

CHAPTER 4

METHOD AND RESEARCH DESIGN

This chapter describes the design of a software verification tool to detect vulnerabilities in UAVs and the methodology used during the development process. In particular, this chapter details the design decisions taken while constructing different tool components. The chapter begins by providing a rationale for using these techniques. This construction was implemented using a V-methodology for software development life cycle [93]. The chapter also addresses any assumptions in creating this solution.

4.1 Project Rationale

The rationale behind this research is precisely to develop and investigate the automatic detection of security vulnerabilities in UAV real-world software through an integration of the fuzzing techniques and ESBMCs in *k*-induction as a novel technique, to overcome obstacles for individual procedures and simultaneously guarantee that the reliability of software is improved.

4.2 Development Methodology

The skeleton of a system is the software architecture and design. It establishes how the system conforms in terms of multiple functional and non-functional requirements [94]. In order to attain quality in the system, this research is based on the V technique [95] used as a software development strategy and its sprints.

4.2.1 V-Model Technique

The V-model is a software development life cycle model in which processes are executed in a sequential V form [95]. It is also known as the verification and validation model as traditional methods were unable to keep pace with the development of software and rapid life changes and also with the development and modification of the requirements of projects. Based on traditional techniques such as waterfall methodology, the V-model takes a long time to complete the project, which involves enormous costs, for redesign and redevelopment in case of incompatibility with system requirements[96].

The first presentation was made by NASA at the NCOSE Symposium in Chattanooga, Tennessee in 1991 [93]. The model presents the sequence stages in the life cycle of the software development and explains the operations to be carried out and the outcomes to be obtained during the project life cycle. The left side of the V-model demonstrates the requirements and creates system configurations, whereas the right side relates to the integration procedure of parts and verifies their validity and efficiency by separately checking each step before manufacturing starts on a large scale [95], as illustrated in Figure 4.1.

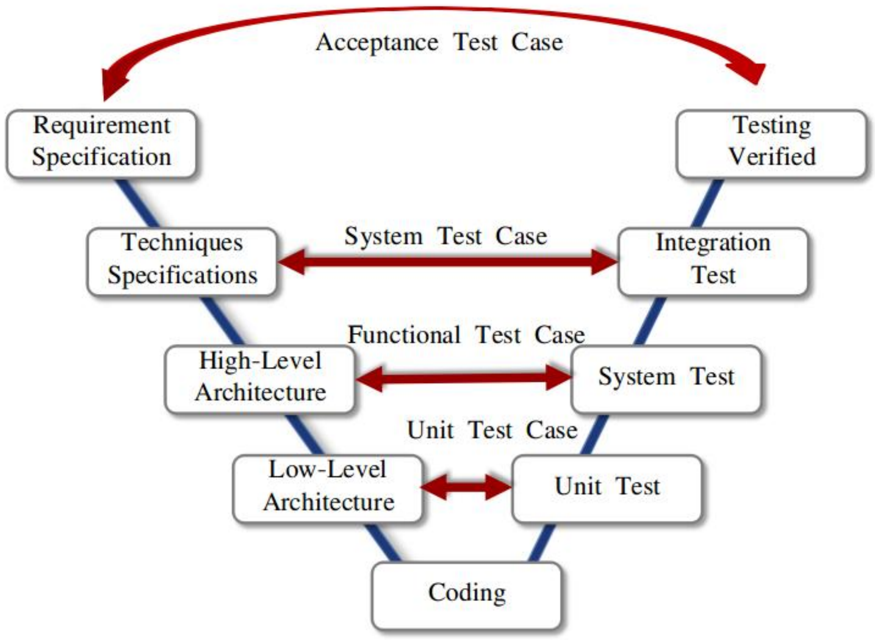


Figure 4.1: Software Development Life Cycle

4.2.1.1 V-Model's Design Phase

1. Requirement specification: Following comprehensive research, this stage started to determine the significance of combining three techniques into one (single) tool. The tools and libraries required for this tool were then fitted.

2. Techniques specifications: After establishing the tool's requirements, the following techniques were identified:

   (a) Abstract interpretation

   (b) Bounded model checking

   (c) Fuzz testing.

3. High-level architecture design: In this phase, an external tool architecture was developed, which includes the three techniques mentioned and which clarifies the data flow between different techniques.

4. Low-level architecture design: In this phase, the complete system is divided into smaller modules. The thorough layout of the system parts is clarified, which is can deemed the Low-level architecture.

Regarding the coding are detailed in Chapters 5 and and testing for inspecting criteria is been parallel to every phase.

4.3 DepthK 3.2 Architecture

The system architecture was designed according to the investigation in Chapter 3. The architecture of this research proposed agent for the vulnerabilities security detection of real-world UAV software is based on hybrid verification techniques, each of them responsible for some aspects of the complete verification process. A high-level DepthK 3.2 architecture diagram is illustrated depicted in Figure 4.2 below.

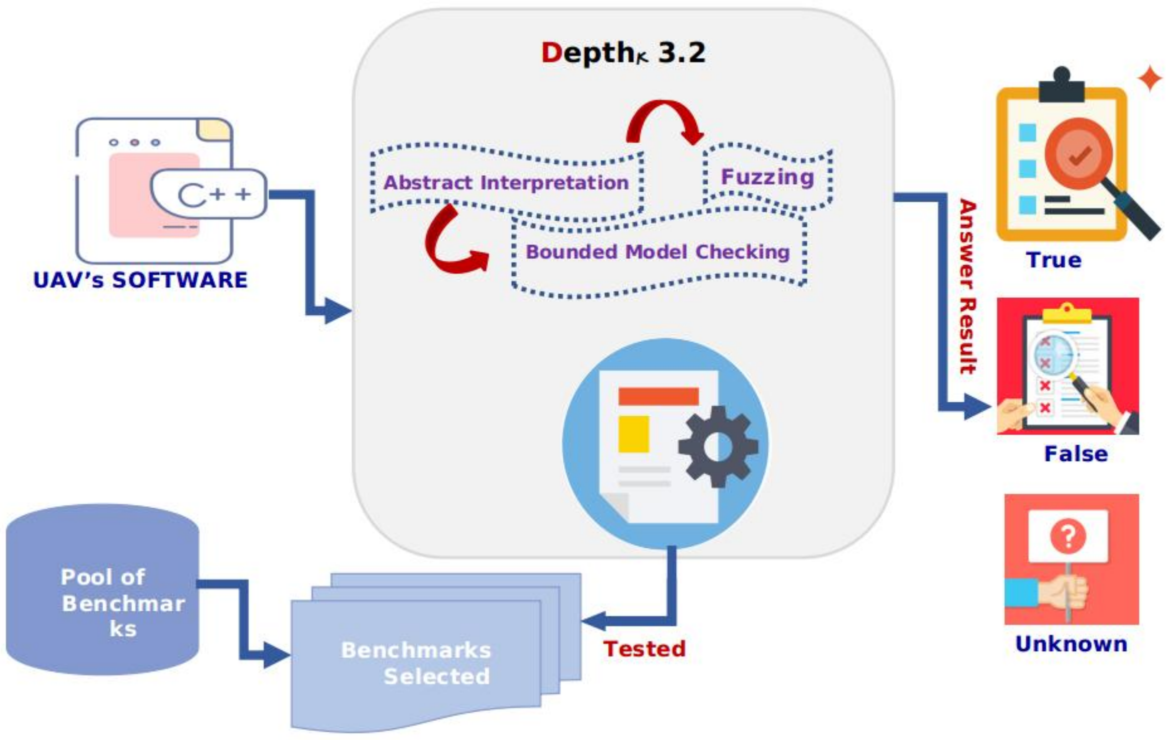


Figure 4.2: High-Level Architecture of UAV's Vulnerability Security Verification Tool.

An abstract flow during the verification phase is outlined below:

Algorithm 1: DepthK 3.2

```
1: procedure DEPTHK 3.2-FUZZ TESTING
2:     Input ← the UAV real-world Binary File
3:     Define ← pre-collected test cases
4:     $errorcounter = 0$
5: loop:
6:     Binary File execute against Define
         if test cases $\neq$ crashing then
7:     goto loop.
         else
8:
         $errorcounter \leftarrow errorcounter++$.
9:     End loop.
         return switch errorcounter do
         case False do  if a vulnerability has been identified ;
         case True do  if the software is free of vulnerabilities ;
         otherwise do  Unknown ;
10: end procedure
11: procedure DEPTHK 3.2
12:     Input ← the UAV real-world C/C++software
13:     Input ← User assertions
14: Compiler:
15:     Parse UAV file
16:     K ← User define
17:     $depth = 0$
18:     $errorcounter \leftarrow errorcounter$
         while  $depth < K$ do
         if Property Hold then
19:     $depth \leftarrow depth++$.
20:     $errorcounter \leftarrow errorcounter++$.
         else
21:
         $depth \leftarrow depth++$.
22:     End While loop
         return switch errorcounter do
         case False do  if a property violation has been identified ;
         case True do  if the specification has been reached ;
         otherwise do  Unknown ;
23: end procedure
```

4.4 Architecture Internal Details

The high-level architecture of the proposed instrument DepthK 3.2 was provided in the previous sections 4.3. This section provides a detailed description of the low-level architecture of this instrument, where each technique is highlighted and explained.

4.4.1 Fuzz Testing Agent

The dynamic part of the verification tool is performed through fuzz testing. Similar to other analytic agents participating in the verification process, the Fuzz test also requires several inputs to check for exceptions such as collisions and memory defects. Figure 4.3 illustrates the architect architectural diagram of fuzz testing.

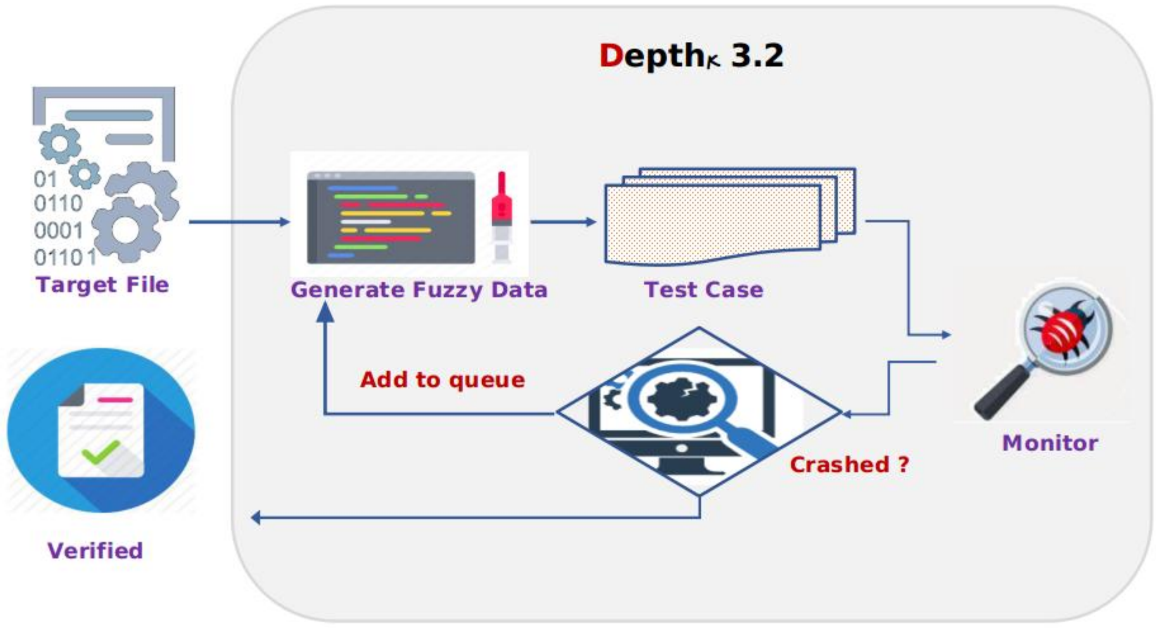


Figure 4.3: Fuzz Agent Architecture

The activities conducted by the fuzz agent are outlined in the following points:

1. ***Identify target software***: The software application, which is going to be tested is marked. As this research focuses on UAV software, UAV software is targeted to this tool.

2. ***Define inputs***: Once the objective software is selected, based on these data, different forms of fuzzers data can be used. These random inputs are generated for testing purposes.

3. ***Produce fuzzy data***: Once the random inputs have been obtained (i.e. unexpected and invalid), as explained in Chapter 2 in Section 2.5, these data are generated in one of two ways:

   - Mutation-based fuzzers, where valid data are collected and modified.

   - Generation-based fuzzers, where data are structured from scratch depending on its particular structure.

   These random test cases are utilised for the software application as inputs.

4. ***Conduct the test case***: The fuzzed data testing process is now performed against the target software. In this phase, the software code is effectively performed with the previous random input.

5. ***Compatibility software monitoring***: Once the software application has been executed, it will be used to crash or other exceptions such as memory disclosures. Under the random input, the software conduct is monitored.

6. ***Determine whether to crash or not***: Following each fuzz test, the request ends, and its initial state is restored. Once a crash has been found, the process will determine if the vulnerability is vulnerable to attack.

4.4.2 Abstract Interpretation Agent

Abstract interpretation agent is the static analysis technique for the present research tool. It is mainly responsible for carrying out the static analysis component of this tool. It is already based on many libraries, which are explained in further detail in this section . Abstract interpretation agent takes the C++ source code as input and safety probabilities and then generates SMT information which is expressed as the logical first-order formulas.

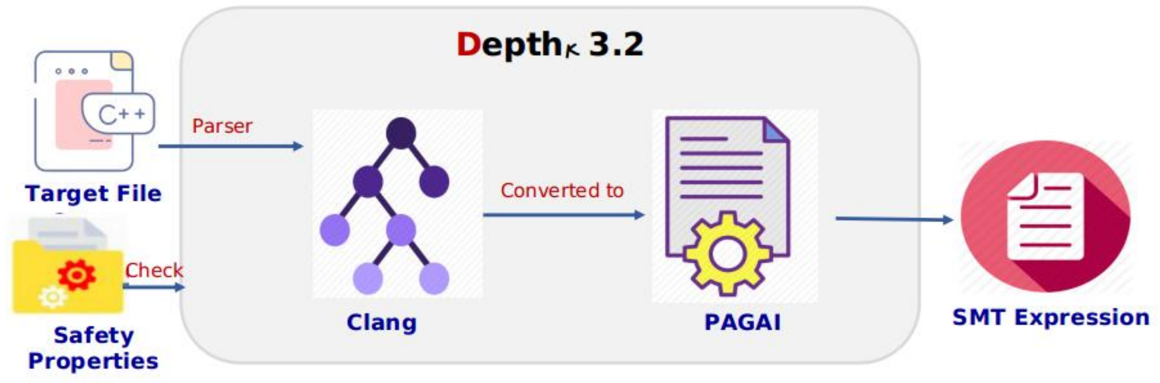


Figure 4.4: Low-Level Architecture of Abstract Interpretation.

The compiler is supplied with a high-level source program, which it converts into an intermediate representation (IR), and then uses a sequence of optimisations, beginning with classical architecture-independent global optimisations, followed by architecture-dependent optimisations, such as registry allocation and instruction planning. These optimisation's typically occur in several passes, in which each pass is optimised in a certain way. Translation validation presents evidence of the accuracy of each optimisation pass, where the positive validation leads to a proof-script and a failed validation results in a counter-example [97].

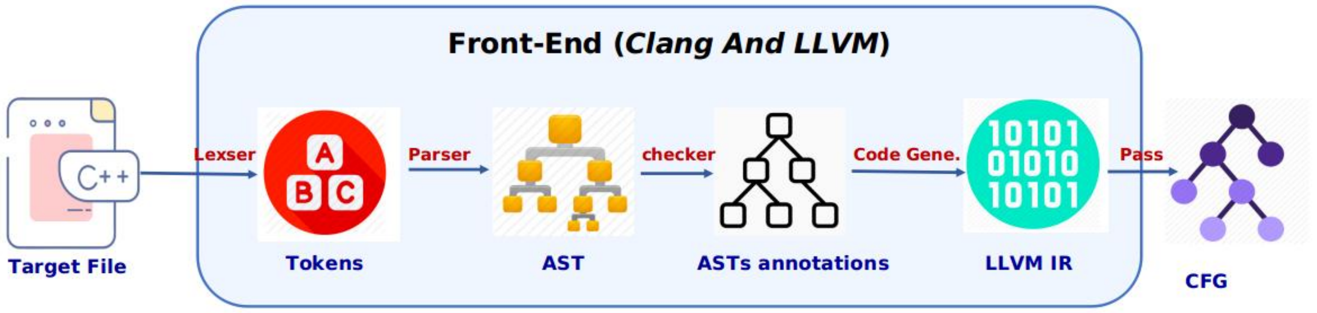


Figure 4.5: Clang's Compiler Architecture

Clang[33] is a C / C++/Objective-C open-source compiler written by STL. Clang is intended to replace GCC as a drop-in replacement. It works as front-end for Low-level virtual machine (LLVM) [1] compiler framework. The primary design objective was to provide clear and expressive user-friendly errors messages. Clang prints the source code to where the parser has run into a problem with a marker. It also has a high-level interface for accessing the AST, caching it and crossing it over various indexing files.

[1]https://llvm.org/

- **Tokens**: In the lexical analysis phase of a compiler, the task to read and divide source code into tokens as a character file. Tokens are like words in a natural language; each of them is a sequence of characters, which represents a unit of information such as if and while in the source program [98].

- **Abstract Syntax Tree**: The parser obtains the sourcecode as tokens from the scanner and analyses the syntax to determine the program structure. This is slimier to do a natural language sentence grammatical analysis. Syntax analysis determines both the program structural elements and its relationships, its results are typically displayed as a syntax tree [98].

- **AST's Annotations**: The program semantic determines its run time behaviour by its features, which include declarations and type-checks. The additional pieces of information such as data type, which are computed with the semantic analyser, are called attributes that annotations have been added to the tree [98].

- **LLVM IR**[2]: The Code generation takes intermediate representation (IR) forward to the optimisation phase and generates target machine code [98]. LLVM can provide an entire compiler scheme center levels, by using intermediate representation code from a compiler and generate an optimised IR. A machine-dependent assembly language code for a target platform can then convert the resulting IR and connected it into CFG [99].

- **CFG**: The Control flow graph displays all paths navigation during execution. Nodes involve basic components of all instructions performed during code execution. Cutting edges describe control flows between the basic nodes. Predecessors are referred to as the adjacent block that can flow to the basic node. Adjacent blocks that can flow to a certain basic node are known as the successors of the control node. The program control will be moved to the next block at the end of each node, usually according to a condition leading to a move to another node [100].

PAGAI [3] [101] is a new, completely automated static analysis tool. PAGAI inputs the LLVM intermediate representation based on the CFG. PAGAI verifies safety properties supplied by users by assertions who used the C/C++ standard.

SMT Expression The Satisfaction module theory (SMT) in the bounded model checking is an extension to the Boolean satisfiability Problem SAT. A problem is satisfiable in the SAT ap-

[2]https://llvm.org/
[3]https://pagai.gricad-pages.univ-grenoble-alpes.fr/about.html

proach if there is an interpretation, which meets the corresponding Boolean formula. The SMT is expressed in logical formulas of the first-order. First-order logic is a generative grammar, which expresses a system in terms of variables (like, X, Y), predicates (for example, x $\geq$ y), connectors ($\wedge$, $\vee$, $\neg$) and quantifiers ($\forall$, $\exists$) [100].

4.4.3 Bounded Model Checker Agent

Once the ESBMC static analysis is completed, the bound model checker agent takes control which is further broken down into several steps.

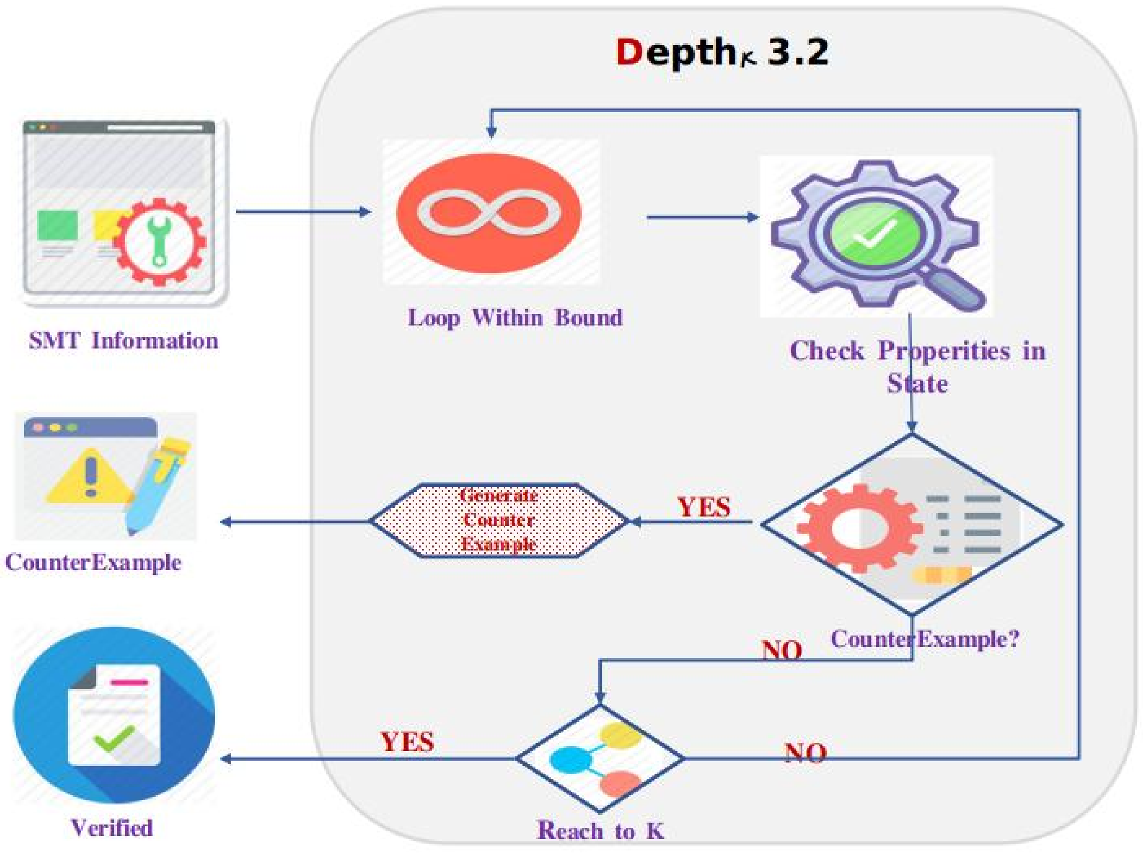


Figure 4.6: Low-Level Of Bounded Model Checker Architecture

1. The source code in the abstract interpretation phase is preprocessed by clang.

2. ESBMC executes the GOTO program symbolically: Loop constructs can be formulated

by means of *while* statements, recursive functions by call and goto statements. GOTO loops are unrolling by doubling loop body K times. An unrolling assertion ensures that the program never needs any further iteration.

3. The program is then converted to a form of SSA. SSA is an IR property requiring every variable to be allocated exactly once and every variable to be defined before being used. In the original IR, the existing variables will be divided into versions and new variables are typically stated by the original name with a subscription in textbooks. Each definition will then receive its own version. Use-def chains are explicit in SSA form, each containing a single component [100].

4. The operation generates two equations of the bit-vector: where C is the set of the constraints and P is the set of the property. BMC transforms $C \vee \neg P$ to a quantifier-free formula to inspect the property.

5. Checking the property needs back-end solvers: When the formula is SAT, a bug is contained in the program, then ESBMC will then produce a counterexample of assignments, which lead to the property violation.

4.4.4 Benchmark Selected

Benchmarking empirical assessment of verification tools is a popular technique in the studies of software verification [102]. In Computer Science, benchmarks were used to compare the efficiency of computer systems, information retrieval algorithms, databases and many other techniques. By using benchmarks, the tools and techniques developed are improved globally [103]. It defines as a standard of measurement or evaluation test or test set for the comparison of the performance of alternative techniques or tools [104], [103]. These tests should be a representative sample of the tasks, which will be solved in real practice by the tool or technique. As the whole population of the problem domain can not be included, a number of tasks serve as substitutes. However, Lu et al. [104] claim that there exists no generally accepted benchmark suite for evaluation of existing or newly proposed techniques in the field of software bug detection. For this, they identified the following benchmark for bug detection selection criteria in their research:

1. ***Representative***, Benchmark should be able to display real buggy apps.

2. ***Diverse***, Benchmark apps should be different with some significant features, including bug types, within the state space.

3. ***Portable***, the Benchmark should be able to assess tools developed on various architectural platforms.

4. ***Accessible***, Benchmark suites are most useful if everyone can access and use them in assessment easily.

5. ***Fair***, which means that the benchmark should not bias toward any detection tool.

This study is based on SV-COMP, which is one of the primary projects aimed at evaluating new software verification, techniques and tools according to Dirk Beyer [105]. The results of the verification run are triple (answer, witness, time) to verify whether the software meets its specification, according to the present SV-COMP regulations [102]. The answer is one of the following results:

- ***TRUE***, if the specification has been met.

- ***FALSE***, if the tool fails to meet the specification.

- ***UNKNOWN***, when the tool is unable to determine the problem or ends with a crash, time-out, or out of memory.

Further elaboration of this benchmark is provided in chapter 6 where it has been used to compare the research tool with similar other ones.

CHAPTER 5

IMPLEMENTATION

The empirical architecture in Chapter 4 was adopted for the implementation of DepthK 3.2 tool. The implementing is here conducted based on the guidelines proposed by Fox and Jennings [106]. It begins by describing the strategies to its plan. Following are the steps are take to set-up the development environment. This discussion includes details of implementing the main components of the tool and highlights important execution characteristics.

5.1 Implementation Plan

The implementation was intended to combine concepts inspired by the techniques learned during a literature review presented in Chapters 2 and 3. These include the abstract interpretation, bounded model checking, and fuzzy testing techniques.

A roadmap for DepthK 3.2 development was established with an implementation plan. Under this plan, the first step was to set up the developer environment on the machine for development purposes. The main emphasis was then on recognising the DepthK 3.2 implementation architecture and identifying its features for this implementation. Finally, the progressive V-model development described in the design stage covered by Chapter 4 was to be completed.

5.1.1 Configuration Environment

The configuration of the setting was performed according to the guidelines provided by DepthK 3.1[1] instructions. The development environment was established by the following steps:

- **Python**: The choice of a suitable programming language can be of crucial significance as it specifies the programming process efficiency and quality; it has significant impacts on final software performance. As the system can be made using many programming languages, the vital criteria for selecting the appropriate language can be characterised as follows.
    1. It contains a number of libraries, which can support the verification function of research techniques.
    2. It must be easy file handling and support for data structures.

  Finally, Python (3.6.8) has been chosen to be the programming language for the development, which meets all the basic requirements of this project.
- **Java**: The system has installed Java 8 Development Kit (JDK) with access to Java the libraries, packages, and APIs used in the development process.

5.1.1.1 Development Environment

In order to facilitate the development of this tool, some ready-made tools and compilers were used for flexibility. These equipment tools have been developed so that developers can use them in their projects instead of writing them from scratch. Table 5.1 lists these utilities, user version, and installation command.

Table 5.1: Development Environment Tools

COMPONENTS	**VERSION**	**COMMAND**
Pycparser	v2.10	`sudo apt-get install python-pycparser`
Clang	v3.5	`sudo apt-get install clang-3.5`
Ctags	v5.8	`sudo apt-get install exuberant-ctags`
GCC compiler		`sudo apt-get install gcc`

[1] Available at https://github.com/omaralhawi/depthk

Following these steps, the environment was prepared for the implementation of DepthK 3.2. The details of this implementation are provided in the other sections of this chapter.

5.2 The Back-End Structure

The integrated analysis technique presented in this research is implemented as a prototype in the security verification tool based on the tool architecture presented in Chapter 4. The Python programming language is used to implement the tool in the form of independent agent-based subsystems as illustrated previously in the tool's architecture.

All agents of the current tool are written and developed in the Python programming language. The tool is implemented in two phases, based on the concept used in the proposed techniques of integrated dynamic and static analysis. Dynamic analysis based on fuzz testing was conducted during the first implementation phase. The second phase concerns with the implementation of the static analysis, which involves subsystems which communicate with each other and use the output of one or more subsystems as its input. The result is based on the output and its format generated during this phase.

The remainder of this chapter discusses more details on the implementation of the complete DepthK 3.2.

5.2.1 Implementing Abstract Interpretation and Bounded Model Checker

Pre-processing (i.e. lexical analysis and parsing, is typically necessary for the implementation of any technique of static code analysis based on abstract interpretation, as mentioned in Section 4.4.2, and the manual building of lexers and parsers is usually uncommon. Therefore, one or more tools are employed for automatic lexer and parser construction, depending on the technology under development.

Since the research present tool, based on version 3.1 of Depthk, then the pre-processing processes were implemented through ESBMC. The technique of static analysis includes first compiler phases, pycparser (v2.10) currently used to parse a C program into an AST to this purpose. *pycparser* is a complete C language parsed by using PLY Library, written with Python. It compiles C source code into a model as an AST. Figure 5.1 and Table 5.2 provide examples of certain modules supplied by the Python's Pycparser, which are ready-to-use.

Parser Function.py

```
def parse(self, text, filename=' ', debuglevel=0):
        """ Parses C code and returns an AST.
            text:
                A string containing the C source code
            filename:
                Name of the file being parsed (for meaningful
                error messages)
            debuglevel:
                Debug level to yacc
        """
        self.clex.filename = filename
        self.clex.reset_lineno()
        self._scope_stack = [dict()]
        self._last_yielded_token = None
        return self.cparser.parse(
                input=text,
                lexer=self.clex,
                debug=debuglevel)
```

Figure 5.1: An Example Code for pycparser Module

Table 5.2: Some Example of Pycparsers' Modules

Pycparsers' Modules [2]	Used
sys	The sys module generally provides the Python interpreter with information on constants, functions and methods.
ast	Python applications can be processed in the Python abstract syntax grammar using the ast module.
pprint	The Pprint module allows the arbitrary data structures in Python to be "pretty-printed," in a form that can be used as an input for the interpreter.

Depending on the present token, *pycparser* requires tokens generated by the ctag and produces an AST node. Implementing the pycparer takes three main phases of the preparation of an AST: a preparation phase, generation of nodes and AST's annotations phase. All these phases are important to ensure that the pycparser can carry out the generated AST. Operations such as `if-else` condition, `goto` and `switch` involve running to jump and ignore other instructions depending on a given condition. These directions are used by pycparer to provide unstructured execution, which jumps between separate nodes all through the AST translation. For example, in CFG a loop is established by setting up `if-else` and `goto` together in a manner that allows a set of compiled code to jump back and forwards. It produces a source code definition index, which is used to find the definitions immediately. Clang[3] is used for compiling the C file and convert it to LLVM bitcode. The LLVM result used to generate the program invariants which be input to PAGAI.

ESBMC is used for *k*-induction verification and CPAchecker[4] for validation of witnesses.where DepthK uses the validators to check results connected to the forward situation and inductive step. Microsoft Research Z3 is used as SMT-Solver. Z3 can be used to verify that logical formulas are satisfactory through one or more theories. It is a low-level tool that is often used as an aspect in other tools which require the solution of logical formulas. The input format of it is an extension to the normal SMT-LIB 2.0. A Z3 script presented as a command sequence. [35].

[3]http://clang.llvm.org

[4]https://cpachecker.sosy-lab.org/

5.2.2 Implementing Fuzz Testing Agent

The procedure outlined in Chapter 4 was implemented to construct this agent. The implementation of this agent was further divided into two sub-phase; pre-fuzz and fuzzing being phase.

In the pre-fuzz phase, the standard library Python offers good logging capabilities for the module logging. Logging requirements depend on the application and may change in the life cycle of the application. Therefore, a logging system must be flexible. Class `fuzzing.LoggerFactory` launches the configuration file of YAML and triggers the logging system.

In addition it defines as a Singleton. Singleton classes are defined by the principle that no more than one example will apply. Figure 5.2 provides fuzz Singleton's function declaration.

logging Singleton.py

```python
import logging
import logging.config
@singleton
class LoggerFactory(object):
    def __init__(self, package_name='fuzzing',
      ↪  config_file='resources/log_config.yaml'):
        self.package_name = package_name
        self.config_file = config_file
        self.config = None

    def initialize(self):
        self.config = self.__read_configuration()
        logging.config.dictConfig(self.config)

    @staticmethod
    def get_instance(identifier):
        logger = logging.getLogger(identifier)
        return logger

    def __read_configuration(self):

        cfg = pkgutil.get_data(self.package_name, self.config_file)
        conf_dict = yaml.load(cfg)
        return conf_dict
```

Figure 5.2: Singleton's Logging

fuzz agent receives UAV file as a binary format after the initialisation step which implemented in pre-phase was completed. The next step is to generate a collection of random module fuzzed test cases. It is a challenge to find suitable test cases to be used with UAV software, due to fuzz testing is a new area for UAV software. The generates random data through the fuzzer function. This tool's research used the generational approach to the test cases with pdfcrack files. Figure 5.3 presents a fuzzer function, where a binary buffer can include files such as, a PDF, an image, etc.

Random Generation Function.py

```python
def fuzzer(buffer, fuzz_factor=101):

    buf = deepcopy(buffer)
    num_writes = random.randrange(math.ceil((float(len(buf)) / fuzz_factor))) +
    ↪  1
    for _ in range(num_writes):
        random_byte = random.randrange(256)
        random_position = random.randrange(len(buf))
        buf[random_position] = random_byte
    return buf
```

Figure 5.3: Test Random Data Generation

The most significant model used in FuzzingLib is shown in Table 5.3.

Table 5.3: Fuzz' Modules by FuzzingLib

Fuzz' Modules	**Used**
random	This module installs pseudo-random number generators for different datasets.
time	This module provides several features related to time.
math	This module provides access to the C standard defined mathematical functions.

The `FuzzExecutor.stat` is a `TestStatCounter` instance. It provides for each implementation the number of successful and unsuccessful runs. In order to combine several test statistics, `TestStatCounter` implements:

Execution Class in FuzzingLib.py

```python
class FuzzExecutor(object):


    def setNotActive(self):
        self.isActive = False

    def __init__(self, app_list, file_list):

        self.logger = logging.getLogger('fuzzing.fuzzer.FuzzExecutor')
        self.logger.info('Initializing FuzzExecutor ...')
        self.apps, self.args = FuzzExecutor.__parse_app_list(app_list)
        self.file_list = file_list
        self.fuzz_factor = 251
        keys = [os.path.basename(app) for app in self.apps]
        self.stats_ = TestStatCounter(keys)
        self.isActive = True

    def run_test(self, runs):

        startingTime = time.time()
        self.logger.info('Start fuzzing ...')
        for _ in range(runs):
            if not self.isActive:
                break
            app = random.choice(self.apps)
            data_file = random.choice(self.file_list)
            fuzzed_file = self._fuzz_data_file(data_file)
            self._execute(app, fuzzed_file)
        self.logger.info('Fuzzing completed.')
        return startingTime
```

Figure 5.4: Execution Class and its Function

`Status` is an enum class that provides backed test status values:

Status values.py

```python
@enum.unique
class Status(enum.Enum):
    FAILED = 0
    SUCCESS = 1
```

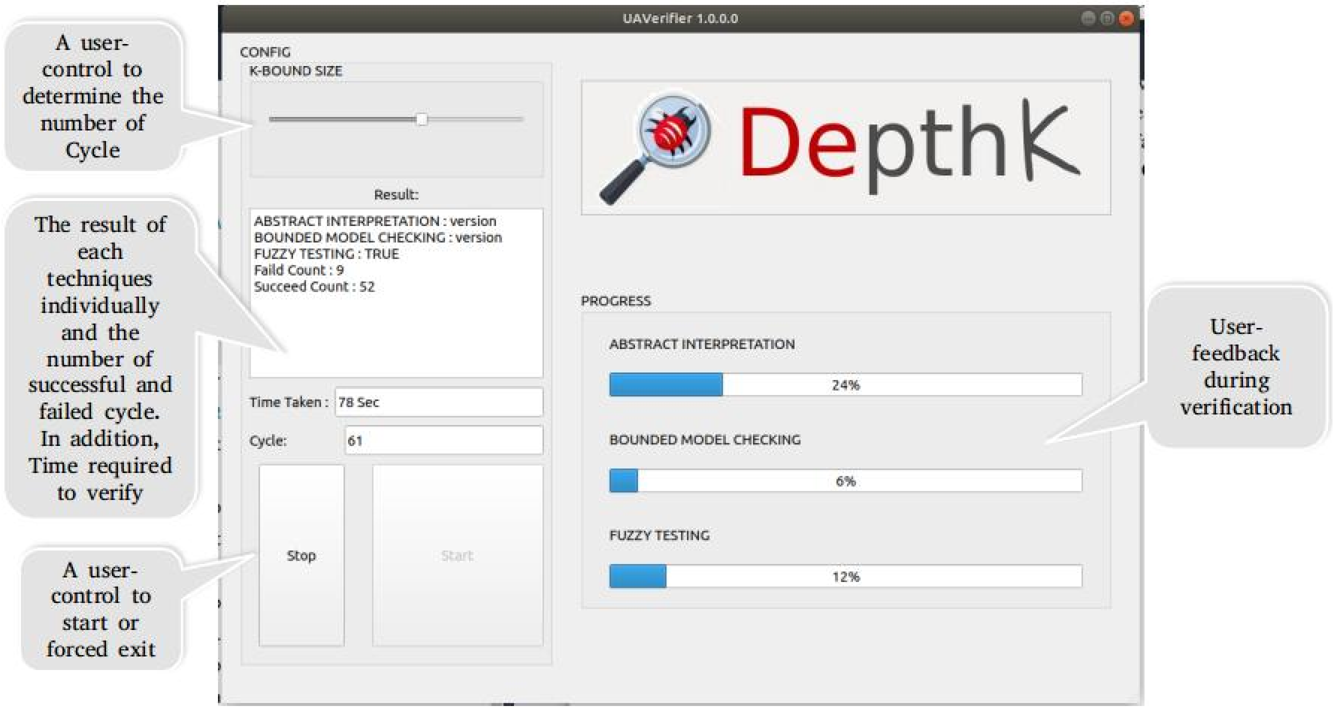
Figure 5.5: Run Status Values

5.3 Display output

Once the verification process has been completed, and the three agents have done its tasks, the tool generates the verification result for the user. It includes the validation result in one of these forms (True/False/Unknown), the cycles' number that user-defined it and time consumed to perform the verification process, where the time unit calculated by the CPU unit.

5.3.1 Graphical User Interface

A user-friendly and simple interface has been created for the user. The graphical user interface (GUI) was built using PyQt [5], which is double-licensed under both a commercial as well as GPL license, depending on Qt[6] library. The prototype was visualised on a one-page screen and the screen was created according to a typical screen layout as illustrated in Figures 5.6 and 5.7. The principal requirement is that the user can use the prototype activities in the least possible procedures and provide feedback on the results for the verification process.


A user-control to determine the number of Cycle
The result of each techniques individually and the number of successful and failed cycle. In addition, Time required to verify
A user-control to start or forced exit
User-feedback during verification
UAVerifier 1.0.0.0
CONFIG
K-BOUND SIZE
DepthK
PROGRESS
Stop
Start

Result:

```
ABSTRACT INTERPRETATION : version
BOUNDED MODEL CHECKING : version
FUZZY TESTING : TRUE
Faild Count : 9
Succeed Count : 52
```

Time Taken : 78 Sec

Cycle: 61

ABSTRACT INTERPRETATION 24%

BOUNDED MODEL CHECKING 6%

FUZZY TESTING 12%

Figure 5.6: User-Interface for the Prototype

[5] https://riverbankcomputing.com/software/pyqt/intro
[6] https://www.qt.io/

information

You Suspend Work !

OK

information

Checking Is Done !

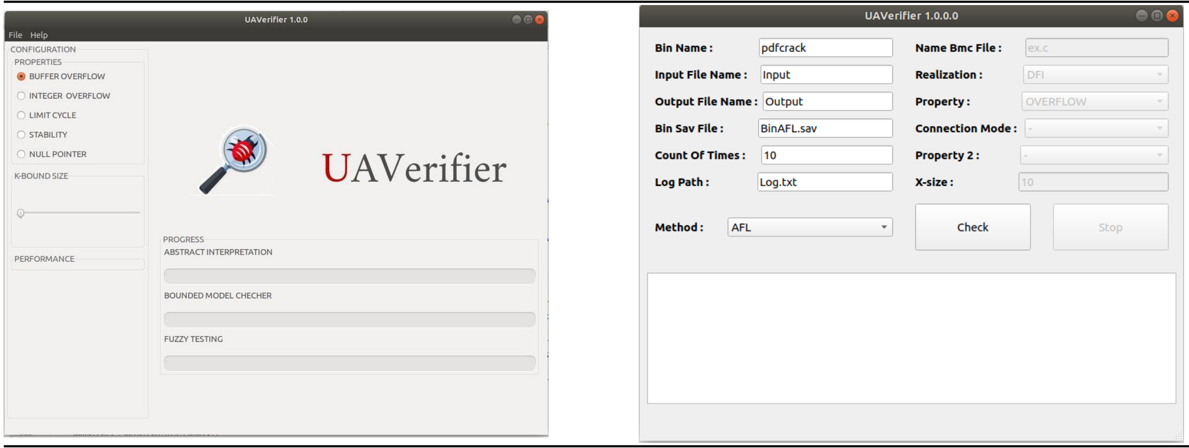
OK

Figure 5.7: User-Message Information

5.4 Implementation Challenges

There were some challenges that we faced during the implementation phase. Initially, it was an attempt to produce a tool that incorporates the three techniques mentioned in the studies purpose 1.4. It has been constructed in C++ based on three libraries: Ikos libraries[7] were used for an abstract interpretation, BMC and AFL libraries were used to implement a fuzz testing. It was also given an initial name as UAVerifier 1.0.0. The challenge was the difference in libraries construction and different compile of the C file for each library.

As a result of the first attempt, a new solution was adopted, which was to construct a tool based on the development of the Dsverifier 2.0.3 tool. Dsverifier 2.0.3 combines the two technologies abstract interpretation and BMC. The new tool was developed to integrate Dsverifier 2.0.3 with AFL libraries. During implementation phase, the research proposal was adopted, which the supervisor Dr Cordeiro had already suggested, because of the problems of the research interface. The study tried to combine Depthk 3.1 with ESBMC v5.0 as in the proposal. However, this attempt did not produce the outcomes intended for in the research.



UAVerifier 1.0.0

File Help

CONFIGURATION

PROPERTIES

[✓ BUFFER OVERFLOW]

INTEGER OVERFLOW

LIMIT CYCLE

STABILITY

NULL POINTER

K-BOUND SIZE

PERFORMANCE

UAVerifier

PROGRESS

ABSTRACT INTERPRETATION

BOUNDED MODEL CHECHER

FUZZY TESTING

UAVerifier 1.0.0.0

Bin Name :	pdfcrack	Name Bmc File :	ex.c
Input File Name :	Input	Realization :	DFI
Output File Name :	Output	Property :	OVERFLOW
Bin Sav File :	BinAFL.sav	Connection Mode :	-
Count Of Times :	10	Property 2 :	-
Log Path :	Log.txt	X-size :	10
Method :	AFL	Check	Stop

Figure 5.8: The GUI Of the Tools,We Have been Trying to Construct Already

[7]https://github.com/NASA-SW-VnV/ikos

CHAPTER 6

EVALUATION AND ANALYSIS OF DIFFERENT VERIFICATION TOOL

This chapter is meant by several assessment metrics to answer research questions ***R2*** and ***R3*** by assessing the efficiency of cutting-edge verification techniques. Evaluation and analysis are conducted based on the method undertaken by Prause et al.[13] and Liang et al. [59], which are published in IEEE 11th International Conference on Software Testing, Verification and Validation. This assessment starts by identifying the evaluation Metrics, which are relevant to the research aims. Additionally, state-of-art tools targeting this security problem evaluated, which are already defined in chapter three.

6.1 Comparing the accuracy and performance of verification tools

The purpose of this section is to evaluate several open-source verification tools in order to assess our tool's efficiency and characteristics than other one. The study has selected three well known publicly available vulnerability detection tools for this assessment. List of tools evaluated, their versions and locations for downloading, are presented in table 6.1.

Table 6.1: Open-Source verification Tools

V.Tool	**VERSION**	**LOCATION**
DSverifer	2.0.3	http://dsverifier.org/
AFL	2.52b	http://lcamtuf.coredump.cx/afl/
IKOS	1.3	https://github.com/NASA-SW-VnV/ikos

The current experiments have been conducted on a computer configured as follows: Intel Core i5-8250U 1.60 GHz processor, 8 GB of RAM, and Ubuntu 18.04 64-bits OS. All execution times submitted are CPU times.

6.1.1 Evaluation Metrics

This research has identified metrics based on the cumulative True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN) score. To identify these metrics in terms of vulnerability detection, the following is further described:
True Positive which means that there is a vulnerability, and this vulnerability is reported in the source code by the tool.
True Negative , which means that there is no vulnerability in the source code, and the tool does not report about any vulnerability.
False Positive , which means that there is no vulnerability in the source code, and vulnerability is reported by the tool.
False Negative , which means that there is a vulnerability in the source code, and that vulnerability is not reported by the tool.

With the resulting two to two-cell confusion matrix, quality criteria for each tool can be determined as ***sensitivity (S)*** and ***precision (P)*** as the following:

$$S = \frac{TP}{TP+FN} \tag{6.1}$$

$$P = \frac{TP}{TP + FP} \tag{6.2}$$

Where sensitivity is the actual fault ratio identified, while the correct reports are precision [13]. Precision is essential measures to know how well the tool is able to exclude false-positive ratios. The precision measurement seeks to assess how much trust can be placed in the tool. The higher the precision value indicates that more confidence can be provided when the tool reports vulnerabilities [107]. However, In this case, ***the Jaccard coefficient (J)***, which is a measure of overlap between actual and reported defects, is a more appropriate qualitative measure because it is irrespective of true negatives [13] :

$$J = \frac{TP}{FP + FN + TP} \tag{6.3}$$

The balance between these false positives and false negatives specified the tools to be sound or complete. These tools are deemed to be sound when they do not produce false negatives, and when the tools do not produce false positives, they are called complete tools [107].

6.1.2 Evaluation Result

Table 6.2: Time and Warnings Generated by Each Tool

Metrics	**TIME (min:sec.csec)**				**WARNING COUNT**			
	DSVerifier	AFL	IKOS	DepthK	DSVerifier	AFL	IKOS	DepthK
True Positive (TP)	V1	V2	V3	V4	V1	V2	V3	V4
True Negative (TN)	V1	V2	V3	V4	V1	V2	V3	V4
False Positive (FP)	V1	V2	V3	V4	V1	V2	V3	V4
False Negative (FN)	V1	V2	V3	V4	V1	V2	V3	V4

Table 6.3: Warning Counts for the Software Verification Tools

V.Tool	**SENSITIVITY**	**PRECISION**	**J.COEFFICIENT**
DSverifer			
AFL			
IKOS			
DepthK			

Figure 6.1: Histogram of percentage of each tool's warnings

6.1.3 Security Vulnerability coverage

Table 6.4: Coverage Comparison of DepthK 3.2 With Other Tools

BENCHMARKS TYPE	**DSverifer**	**AFL**	**IKOS**	**DepthK**
MemSafety		✓		
Overflows	⊗	⊗		
ReachSafety	✓		✓	
ConcurrencySafety				

6.2 Performance Evaluation and Discussion

Table 6.5: Performance Evaluation for Open-Source Verification Tools

V.Tool	**ADVANTAGE**	**DISADVANTAGE**	**DETECT LEVEL**
DSverifer			
AFL			
IKOS			
DepthK			

You should also describe your benchmarks here. Where did you collect them? Which UAV models are they? please provide as much information as you can about the employed benchmarks.

Rutar et al.[108] compare tools

CHAPTER 7

CONCLUSIONS

7.1 Conclusions Summary

The demand for the development of specialised applications, including UAV applications [12], has significantly increased . However, these applications are highly vulnerable if they are not validated before publication. Different techniques and tools for software validation have been effectively constructed and developed for this purpose. However, the safety inspection process still requires a necessary level of automation.

This study, implemented a technique of static analyses centred on abstract interpretation and bounded model checker that was designed in order to incorporate such techniques with the dynamic analysis technique based on fuzz testing. These guidelines allow for static analysis to detect prospective causes of deficiencies. A dynamic analytical technique for working with static analytical technique was also created and enhanced through this study. Dynamic analytical technology provides a double checker and helps to avoid possible application attacks due to unexpected inputs. In order to demonstrate our this technique as robustly and effective, the researchers tested the tool with other current tools and found that the tool performs quite well in comparison. The tool improves the coverage of the number and types of vulnerabilities compared to other tools.

7.2 Limitations

This project's work is restricted by a number of constraints, despite its mainly focus on the method and its effectiveness in identifying multiple type of vulnerabilities detected in the UAV software. The limitations of this research includee the following things:

1. Many software verification tools are already accessible on the market, some of which are a business products which require payment to use. This research is, therefore focused on open source tools, which may be used and modified by the general public.

2. Some of software verification tools support numerous languages, while others are language-specific. This research focuses primarily on open-source tools related to C, C++ and/or Python. While the research includes a theoretical assessment of tools that support these languages, it only conducts a practical assessment of C, C++ analysis tools due to the restricted time of the study.

3. Since the tool presented in this research is based on defined k as depth length , the effectiveness of the tool relies entirely on the k length. If the length is large, then the tool significantly enhances efficiency, and the converse is true for tiny lengths.

7.3 Future Work

Although the integrated technique employing fuzzing, bounded model checking, and abstract interpretation, has enhanced the system's reliability over other closely related projects, future evaluations of this tool should focus more efficiently on detecting further vulnerabilities and should support other languages. This research proposed that artificial neural networks be used rather than bounded model checking in order to bring the value of unrestricted system verification to a predefined k-length, and thereby improves verification effectiveness in large systems.

Artificial neural networks (ANNs) can be defined as structures consisting of densely connected, simple adaptive processing elements (known as artificial neurons or nodes) which can be used to conduct massive parallel for data processing and data representation computations [109]. The attraction of ANNs is based on the extraordinary characteristics of the biological system data processing such as non-linearity, high Parallel capability, robustness, fault and failure tolerance, learning, ability to of working with incomplete knowledge and fuzz data and capacity to generalise data [109].

Systems verification typically involves computing either the number of states reachable from the initial states or specific fixed points related to logical formulae. Instead of calculating these states by using the transition relationship iteratively, it will view as a target to be learned by responding to specific questions.The primary concept of use ANNs, it are supported by a description of the system model that must be evaluated in terms of its variable declarations, initial state and the property that are to be verified. The result is whether or not the system meets the property. In addition, a false response will be applied to provide a counterexample of the property violation.

This proposal is expected to be appropriate for extremely large systems, decreasing the time needed to calculate reach states and their capacity to detect fuzz data without the need for verifying data generation techniques. In addition, more than one programming language will be supported.

Bibliography

[1] D. G. Bell and G. P. Brat, "Automated software verification validation: An emerging approach for ground operations," in *2008 IEEE Aerospace Conference*, pp. 1–8, March 2008.

[2] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, p. 6, 2018.

[3] L. Cordeiro, "Automated software verification and synthesis in unmanned aerial vehicles." `https://ssvlab.github.io/lucasccordeiro/talks/dts2018_slides.pdf`, 2019.

[4] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "Esbmc 5.0: an industrial-strength c model checker," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 888–891, ACM, 2018.

[5] A. Rehn, "Matching source-level cfg basic blocks to llvm ir basic blocks." `https://adamrehn.com/articles/matching-cfg-blocks-to-basic-blocks/`, 2019.

[6] E. Jääskelä, "Genetic algorithm in code coverage guided fuzz testing," *Dept. Comput. Sci. Eng., Univ. Oulu*, 2016.

[7] M. Alvarez, N. Bradley, P. Cobb, S. Craig, R. Iffert, L. Kessem, J. Kravitz, D. McMillen, and S. Moore, "Ibm x-force threat intelligence index 2017 the year of the mega breach," *IBM Security,(March)*, pp. 1–30, 2017.

[8] P. Inc, "Basic integer overflows." `https://www.cs.utexas.edu/~shmat/courses/cs361s/blexim.txt`, 2019.

[9] CWE, "Divide by zero." `https://cwe.mitre.org/data/definitions/369.html`, 2019.

[10] www.tutorialspoint.com, "C++ null pointers." https://www.tutorialspoint.com/cplusplus/cpp_null_pointers, 2019.

[11] A. Mancini, F. Caponetti, A. Monteriu, E. Frontoni, P. Zingaretti, and S. Longhi, "Safe flying for an uav helicopter," in *2007 Mediterranean Conference on Control & Automation*, pp. 1–6, IEEE, 2007.

[12] T. T. E. BOARD, "faa needs to make thoughtful safety rules before drones deliver our packages or pizza 2019." https://www.latimes.com/opinion/editorials/la-ed-adv-drone-delivery-20151218-story.html, 2019.

[13] C. Prause, R. Gerlich, and R. Gerlich, "Evaluating automated software verification tools," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 343–353, IEEE, 2018.

[14] N. R. Council, *Statistical Software Engineering*. National Research Council, 1996.

[15] J. C. Westland, "The cost of errors in software development: evidence from industry," *Journal of Systems and Software*, vol. 62, no. 1, pp. 1–9, 2002.

[16] L. C. Baldor, "Flashy drone strikes raise status of remote pilots," *The Boston Globe*, 2012.

[17] L. R. Humphrey, E. M. Wolff, and U. Topcu, "Formal specification and synthesis of mission plans for unmanned aerial vehicles," in *2014 AAAI Spring Symposium Series*, 2014.

[18] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *International conference on tools and algorithms for the construction and analysis of systems*, pp. 193–207, Springer, 1999.

[19] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in *ACM SIGPLAN Notices*, pp. 196–207, ACM, 2003.

[20] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[21] M. Bishop, *Introduction to computer security*. Addison-Wesley, 2005.

[22] A. Booth, A. Sutton, and D. Papaioannou, *Systematic approaches to a successful literature review*. Sage, 2016.

[23] S. T. March and G. F. Smith, “Design and natural science research on information technology,” *Decision support systems*, vol. 15, no. 4, pp. 251–266, 1995.

[24] O. Grumberg and H. Veith, *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008.

[25] B. Homès, *Fundamentals of software testing*. John Wiley & Sons, 2013.

[26] P. A. Abdulla and K. R. M. Leino, “Tools for software verification,” 2013.

[27] E. M. Clarke and E. A. Emerson, “Design and synthesis of synchronization skeletons using branching time temporal logic,” in *Workshop on Logic of Programs*, pp. 52–71, Springer, 1981.

[28] G. K. Palshikar, “an introduction to model checking 2019,” 2019.

[29] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu, *et al.*, “Bounded model checking.,” *Advances in computers*, vol. 58, no. 11, pp. 117–148, 2003.

[30] B. Wozna, “Bounded model checking for the universal fragment of ctl*,” *Fundamenta Informaticae*, vol. 63, no. 1, pp. 65–87, 2004.

[31] L. Cordeiro, J. Morse, D. Nicole, and B. Fischer, “Context-bounded model checking with esbmc 1.17,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 534–537, Springer, 2012.

[32] L. Cordeiro, B. Fischer, and J. Marques-Silva, “Smt-based bounded model checking for embedded ansi-c software,” *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 957–974, 2011.

[33] C. Compiler, “Clang: a c language family frontend for llvm.”

[34] R. Brummayer and A. Biere, “Boolector: An efficient smt solver for bit-vectors and arrays,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 174–177, Springer, 2009.

[35] L. De Moura and N. Bjørner, “Z3: An efficient smt solver. tools and algorithms for the construction and analysis of systems,(tacas’08), lncs,” 2008.

[36] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani, “The mathsat 4 smt solver,” in *International Conference on Computer Aided Verification*, pp. 299–303, Springer, 2008.

[37] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *International Conference on Computer Aided Verification*, pp. 171–177, Springer, 2011.

[38] B. Dutertre and L. De Moura, "The yices smt solver," *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, vol. 2, no. 2, pp. 1–2, 2006.

[39] P. Cousot and R. Cousot, "Basic concepts of abstract interpretation," in *Building the Information Society*, pp. 359–366, Springer, 2004.

[40] D. Grasso, A. Fantechi, A. Ferrari, C. Becheri, and S. Bacherini, "Model based testing and abstract interpretation in the railway signaling context," in *2010 Third International Conference on Software Testing, Verification and Validation*, pp. 103–106, IEEE, 2010.

[41] P. Cousot, "Formal verification by abstract interpretation," in *NASA Formal Methods Symposium*, pp. 3–7, Springer, 2012.

[42] H. Post, C. Sinz, A. Kaiser, and T. Gorges, "Reducing false positives by combining abstract interpretation and bounded model checking," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 188–197, IEEE, 2008.

[43] O. Bouissou, E. Conquet, P. Cousot, R. Cousot, J. Feret, K. Ghorbal, E. Goubault, D. Lesens, L. Mauborgne, A. Miné, *et al.*, "Space software validation using abstract interpretation," in *Proc. of the Int. Space System Engineering Conf.,Data Systems in Aerospace (DASIA 2009)*, ESA, 2009.

[44] J.-L. Boulanger, *Static analysis of software: The abstract interpretation.* John Wiley & Sons, 2013.

[45] L. Chen, A. Miné, J. Wang, and P. Cousot, "Linear absolute value relation analysis," in *European Symposium on Programming*, pp. 156–175, Springer, 2011.

[46] A. Miné, "The octagon abstract domain," *Higher-order and symbolic computation*, vol. 19, no. 1, pp. 31–100, 2006.

[47] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 84–96, ACM, 1978.

[48] L. Chen, J. Liu, A. Miné, D. Kapur, and J. Wang, "An abstract domain to infer octagonal constraints with absolute value," in *International Static Analysis Symposium*, pp. 101–117, Springer, 2014.

[49] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[50] G.-H. Liu, G. Wu, Z. Tao, J.-M. Shuai, and Z.-C. Tang, "Vulnerability analysis for x86 executables using genetic algorithm and fuzzing," in *2008 Third International Conference on Convergence and Hybrid Information Technology*, vol. 2, pp. 491–497, IEEE, 2008.

[51] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[52] J. Patra and M. Pradel, "Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data," *TU Darmstadt, Department of Computer Science, Tech. Rep. TUD-CS-2016-14664*, 2016.

[53] W. Chunlei, L. Li, and L. Qiang, "Automatic fuzz testing of web service vulnerability," in *2014 International Conference on Information and Communications Technologies (ICT 2014)*, IET, 2014.

[54] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *ACM SIGPLAN Notices*, pp. 283–294, ACM, 2011.

[55] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "A taint based approach for smart fuzzing," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 818–825, IEEE, 2012.

[56] Cwe.mitre.org, "About cwe." https://cwe.mitre.org/about/index.html, 2019.

[57] Cwe.mitre.org, "Cwe list version 3.3." https://cwe.mitre.org/data/index.html, 2019.

[58] G. McGraw, "Software security," *IEEE Security & Privacy*, vol. 2, no. 2, pp. 80–83, 2004.

[59] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[60] "Software failures cost 2019." http://servicevirtualization.com/report-software-failures-cost-1-1-trillion-2016/, 2019.

[61] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, "Finding software vulnerabilities by smart fuzzing," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pp. 427–430, IEEE, 2011.

[62] J. Jurn, T. Kim, and H. Kim, "An automated vulnerability detection and remediation method for software security," *Sustainability*, vol. 10, no. 5, p. 1652, 2018.

[63] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, vol. 2, pp. 119–129, IEEE, 2000.

[64] M. Zitser, R. Lippmann, and T. Leek, "Testing static analysis tools using exploitable buffer overflows from open source code," in *ACM SIGSOFT Software Engineering Notes*, pp. 97–106, ACM, 2004.

[65] E. U. A. for Cybersecurity Published under Glossary, "Buffer overflow." `https://www.enisa.europa.eu/topics/csirts-in-europe/glossary/buffer-overflow`, 2019.

[66] SearchSecurity, "what is buffer overflow?." `https://searchsecurity.techtarget.com/definition/buffer-overflow`, 2019.

[67] S. Tripathi, G. Grieco, and S. Rawat, "Exniffer: Learning to prioritize crashes by assessing the exploitability from memory dump," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 239–248, IEEE, 2017.

[68] G. G. Software, "Famous software bugs." `https://www.gimpel.com/famous_software_bugs.html`, 2019.

[69] I. S. Committee *et al.*, "754-2008 ieee standard for floating-point arithmetic," *IEEE Computer Society Std*, vol. 2008, p. 517, 2008.

[70] S. Keele *et al.*, "Guidelines for performing systematic literature reviews in software engineering," tech. rep., Technical report, Ver. 2.3 EBSE Technical Report. EBSE, 2007.

[71] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in *2012 fourth international conference on multimedia information networking and security*, pp. 152–156, IEEE, 2012.

[72] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 138–157, IEEE, 2016.

[73] P. Godefroid and K. Sen, "Combining model checking and testing," in *Handbook of Model Checking*, pp. 613–649, Springer, 2018.

[74] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated software engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[75] G. Holzmann, "Static source code checking for user-defined properties," in *Proc. IDPT*, vol. 2, 2002.

[76] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, H. Zheng, *et al.*, "Bandera: Extracting finite-state models from java source code," in *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*, pp. 439–448, IEEE, 2000.

[77] W. Rocha, H. Rocha, H. Ismail, L. Cordeiro, and B. Fischer, "Depthk: a k-induction verifier based on invariant inference for c programs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 360–364, Springer, 2017.

[78] M. R. Gadelha, F. Monteiro, L. Cordeiro, and D. Nicole, "Esbmc v6. 0: Verifying c programs using k-induction and invariant inference," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 209–213, Springer, 2019.

[79] A. Deutsch, "Static verification of dynamic properties," in *ACM SIGAda 2003 Conference*, 2003.

[80] V. K. To, T. V. A. Nguyen, and T. T. Nguyen, "Vtse–verification tool based on symbolic execution," tech. rep., VNU-UET, 2018.

[81] D. A. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities.," in *NDSS*, pp. 2000–02, 2000.

[82] D. Vandenbroeck, "how the sdl helped improve security in office 2010 / 2019." `https://blogs.technet.microsoft.com/office2010/2010/05/10/how-the-sdl-helped-improve-security-in-office-2010/`, 2019.

[83] D. Lettnin and M. Winterholer, *Embedded Software Verification and Debugging*. Springer, 2017.

[84] D. R. Dams and K. S. Namjoshi, "Orion: High-precision methods for static error analysis of c and c++ programs," in *International Symposium on Formal Methods for Components and Objects*, pp. 138–160, Springer, 2005.

[85] O. Ponsini, C. Michel, and M. Rueher, “Combining constraint programming and abstract interpretation for value analysis of floating-point programs,” in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 775–776, IEEE, 2012.

[86] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, “Saner: Composing static and dynamic analysis to validate sanitization in web applications,” in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pp. 387–401, May 2008.

[87] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution.,” in *NDSS*, pp. 1–16, 2016.

[88] L. Chaves, I. V. Bessa, H. Ismail, A. B. dos Santos Frutuoso, L. Cordeiro, and E. B. de Lima Filho, “Dsverifier-aided verification applied to attitude control software in unmanned aerial vehicles,” *IEEE Transactions on Reliability*, vol. 67, no. 4, pp. 1420–1441, 2018.

[89] M. Zalewski, “american fuzzy lop.” http://lcamtuf.coredump.cx/afl/, 2019.

[90] J. Fell, “A review of fuzzing tools and methods,” tech. rep., Technical Report. https://dl. packetstormsecurity. net/papers/general/a . . . , 2017.

[91] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, “The astrée analyzer,” in *European Symposium on Programming*, pp. 21–30, Springer, 2005.

[92] “Astrée runtime error analyzer.” https://www.absint.com/astrée/index.htm, 2019.

[93] N. B. Ruparelia, “Software development lifecycle models,” *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 3, pp. 8–13, 2010.

[94] M. Mekni, G. Mounika, C. Sandeep, and B. Gayathri, “Software architecture methodology in agile environments,” *J Inform Tech Softw Eng*, vol. 7, no. 195, p. 2, 2017.

[95] S. Balaji and M. S. Murugaiyan, “Waterfall vs. v-model vs. agile: A comparative study on sdlc,” *International Journal of Information Technology and Business Management*, vol. 2, no. 1, pp. 26–30, 2012.

[96] A. Cline, *Agile development in the real world.* Springer, 2015.

[97] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg, “Voc: A translation validator for optimizing compilers,” *Electronic notes in theoretical computer science*, vol. 65, no. 2, pp. 2–18, 2002.

[98] C. Louden Kenneth, “Compiler construction: Principles and practice,” *Course Technology*, 1997.

[99] T. U. of Illinois, “The llvm compiler infrastructure project.” `https://llvm.org/`.

[100] J. Henry, *Static Analysis by Abstract Interpretation and Decision Procedures*. PhD thesis, Université de Grenoble, 2014.

[101] J. Henry, D. Monniaux, and M. Moy, “Pagai: A path sensitive static analyser,” *Electronic Notes in Theoretical Computer Science*, vol. 289, pp. 15–25, 2012.

[102] L. Cordeiro, D. Kroening, and P. Schrammel, “Benchmarking of java verification tools at the software verification competition (sv-comp),” *arXiv preprint arXiv:1809.03739*, 2018.

[103] S. E. Sim, S. Easterbrook, and R. C. Holt, “Using benchmarking to advance research: a challenge to software engineering,” in *25th International Conference on Software Engineering, 2003. Proceedings.*, pp. 74–83, May 2003.

[104] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, “Bugbench: Benchmarks for evaluating bug detection tools,” in *Workshop on the evaluation of software defect detection tools*, vol. 5, 2005.

[105] D. Beyer, “Software verification with validation of results,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 331–349, Springer, 2017.

[106] B. Hahn Fox and W. G. Jennings, “How to write a methodology and results section for empirical research,” *Journal of Criminal Justice Education*, vol. 25, no. 2, pp. 137–156, 2014.

[107] J. Shrestha, “Static program analysis,” 2013.

[108] N. Rutar, C. B. Almazan, and J. S. Foster, “A comparison of bug finding tools for java,” in *15th International Symposium on Software Reliability Engineering*, pp. 245–256, IEEE, 2004.

[109] I. A. Basheer and M. Hajmeer, “Artificial neural networks: fundamentals, computing, design, and application,” *Journal of microbiological methods*, vol. 43, no. 1, pp. 3–31, 2000.