Finding Software Vulnerabilities in Unmanned Aerial Vehicles

Using Software Verification

A thesis submitted to the University of Manchester for the degree of Master of Philosophy in the Faculty of Science & Engineering

2021

Omar M. K. Alhawi

School of Engineering, Department of Computer Science

Contents

List of Figures **5**

List of Tables **6**

1 Introduction **10**
1.1 Problem Description 11
1.2 Research Objectives and Contributions 12
1.3 Outline of the Solution 14
1.3.1 Investigation of UAV Software & Associated Threats 14
1.3.2 Examination & Development of Existing Verification Techniques . 15
1.4 Organization of the Dissertation 16

2 Background **17**
2.1 UAVs 17
2.1.1 UAV Security 17
2.1.2 UAV Software 18
2.1.3 CIA Threat Summary 19
2.2 Overview of Fuzzing 19
2.2.1 Types of Fuzzers 20
2.2.2 Fuzzing Process 20
2.3 Overview of Model Checking 21
2.4 Bounded Model Checking 22
2.5 Invariant Inference 23
2.6 Related Work 24

3 Research Methodology **27**
3.1 Threats and Vulnerability Identification 27
3.2 Attack Possibilities 27
3.3 The Proposed Verification Methodology 28

3.3.1 Fuzzing . . . 28
3.3.2 Induction-based Verification . . . 29

4 Verification and Refutation of C Programs based on *k*-Induction and Invariant Inference 31
4.1 Introduction . . . 33
4.2 Background . . . 36
4.2.1 Bounded Model Checking . . . 36
4.2.2 Induction-based Verification of C Programs . . . 38
4.2.3 Property-directed Reachability (or IC3) . . . 40
4.3 Induction-based Verification of C Programs Using Invariants . . . 42
4.3.1 The *k*-Induction Algorithm . . . 42
4.3.2 Extended *k*-Induction Algorithm . . . 44
4.3.3 Invariant Generation using PIPS . . . 46
4.3.4 Invariant Generation using PAGAI . . . 47
4.3.5 Illustrative Example . . . 49
4.4 Experimental Evaluation . . . 52
4.4.1 Experimental Setup . . . 53
4.4.2 Experimental Results . . . 55
4.5 Related Work . . . 64
4.6 Conclusions . . . 68

5 Finding Security Vulnerabilities in Unmanned Aerial Vehicles Using Software Verification 70
5.1 Introduction . . . 72
5.1.1 Contributions . . . 73
5.1.2 Organisation . . . 74
5.2 Background . . . 75
5.2.1 Generic Model of UAV Systems . . . 75
5.2.2 Cyber-Threats . . . 75
5.2.3 Verification of Security in UAVs . . . 76
5.3 Finding Software Vulnerabilities in UAVs Using Software Verification . . . 77
5.3.1 Software In-The-Loop . . . 77
5.3.2 Illustrative Example Using UAV swarm . . . 79
5.3.3 Verifying UAV Software using Fuzzing and BMC . . . 80
5.3.4 UAV Communication Channel . . . 82
5.4 Preliminary Experimental Evaluation . . . 84

5.4.1 Description of Benchmarks 84
5.4.2 Objectives 85
5.4.3 Results 85
5.4.4 Threats to Validity 87
5.5 Conclusions and Future Work 87

6 Conclusion and Future Work **89**
6.1 Conclusion 89
6.2 Future Work 90

Bibliography **91**

Word Count: 33844

List of Figures

1.1 Threats investigation using CIA triad 15

2.1 UAV design 17
2.2 Fuzzing Process 21
2.3 Representation of a transition system 22

3.1 Bash script used to fuzz UAVs 29
3.2 The proposed methodology for UAV software verification 30

4.1 Simple unbounded loop program. 33
4.2 Finite unwinding done by BMC. 34
4.3 Sample with PIPS invariants using the structure #init. 46
4.4 Verification example with two properties. 49
4.5 Verification example rewritten during the inductive step. 50
4.6 Verification example with invariants. 51
4.7 Score regarding loops subcategory. 61
4.8 Score regarding embedded systems. 61
4.9 Results for the loops subcategory. 63
4.10 Results for the embedded programs. 63

5.1 UAV types: multi-rotor (a), fixed wing (b), and single-rotor (c). 75
5.2 Functional structure of UAVs. 76
5.3 Python script to connect to a vehicle (real or simulated). 78
5.4 Python code fragment to read and view various data status of Tello UAV. . . 79
5.5 UAV Swarm Competition. 80
5.6 Test case from the Tello UAV embedded software. 82
5.7 GPS-satellite-signal is overlaid by a spoofed GPS-signal. 83

List of Tables

4.1 Experimental results for the Powerstone, SNU, and WCET benchmarks. . . 55
4.2 Experimental results for the SV-COMP'19. 57
4.3 PDR-based experimental results SV-COMP'18. 57

5.1 DepthK Results in SV-COMP 2019. 86
5.2 Fuzzing Approaches Comparison. 86
5.3 Results of the UAV Swarm Competition. 87

Abstract

Unmanned Aerial Vehicles (UAVs) usage in various tasks has increased exponentially in the recent past, which enable users to complete vital missions efficiently and without risking human lives. However, and like any other systems, it poses a cybersecurity threat if not handled correctly. Therefore, it is essential to know the risk and understand the impact of various possible attacks on the overall UAVs. For this purpose, model checking and fuzzing techniques have been applied to improve the UAV approach's security. Nevertheless, there is little effort focused on using those methods in the software domain, especially when investigating low-level implementation vulnerabilities related to UAV software.

In this dissertation, UAV risks and the impact of cyberattacks investigated by applying bounded model checking and fuzzing techniques. As a result, a new verification algorithm was developed in a tool named $DepthK$ to verify UAV software correctness. The work's primary research objectives examined through two scientific papers centred on these key research disciplines. These studies undertaken described in Chapter 4 and 5.

Paper 1 (*Verification and refutation of C programs based on k-induction and invariant inference*) outlines a new verification algorithm for detecting UAV software vulnerabilities (e.g., concurrency and buffer overflow bugs). The approach's effectiveness was evaluated and showed better results compared to the state-of-the-art testing tools. The new framework also formed the basis for UAVs software verification development.

Paper 2 (*Finding Security Vulnerabilities in Unmanned Aerial Vehicles Using Software Verification*) demonstrated the risks from exploiting UAV software vulnerabilities and the effectiveness of the proposed approach. The experiment results show UAV software failures that hardly detected by other verifiers and was quickly and efficiently detected by the proposed verification algorithm.

Declaration

I declare that no portion of the work referred to in the thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright Statement

I The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

II Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

III The ownership of certain Copyright, patents, designs, trademarks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

IV Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=2442 0), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see http://www.library.manchester.ac.uk/about/regulations/) and in The University's policy on Presentation of Theses.

Acknowledgements

I want to thank my family for the love, support, and constant encouragement I have gotten over the years. In particular, I would like to thank my wife for her support during my education journey, and that's why I dedicate my success to all of them.

CHAPTER 1

Introduction

Drones or Unmanned Aerial Vehicles (UAVs) are a system comprising several sub-systems, including the aircraft, its payloads, the Ground Control Station (GCS), aircraft launch, recovery sub-systems and communication sub-systems [9]. Applications based on autonomous UAVs are becoming very usual, both in military and civil missions. According to PricewaterhouseCoopers (PWC), the impacts of the UAVs market on the UK's gross domestic economy are GBP 42 bn, and it will offer 628,000 new jobs in its economy [120]. During the last decades, the UAV approach has been used in various military and civil applications, such as training targets, aerial surveillance, journalism, and entertainment, due to the steady growth in the development of UAV systems, applications, and the continuous development of its embedded systems[105]. In late 2019 and according to Civil Aviation Authority (CAA), commercial UAVs were illegal to fly in the UK airspace without registering and passing a theory test, and limit its use to 400 feet high and away from airports and air traffic [10]. The availability of low-cost commercial UAVs and DIY drones have also promoted recreational use as well as research. Nowadays, constructing a drone is inexpensive, and individuals can purchase separate parts from online stores to assemble. Incorrect software or failure inaccurate navigation may lead to dangerous and life-threatening accidents and thus, necessitates prior testing of the software in a simulation environment [92, 90, 140]. The failure of the control software of the Arian-5 rocket and the Mars rover are stern reminders of what can happen when systems do not perform as per specifications [98].

Embedded systems, such as those in UAVs, impose several restrictions (*e.g.*, response time and data accuracy) that must be met and measured, according to users requirements; otherwise, failures may lead to catastrophic situations. For example, safety risks and safety concerns to large aeroplanes and ground installations which widespread inhibiting uptake of drones. For instance, Gatwick airport incident in the U.K. late 2018 and the RQ-170 UAV accident in 2012, where Iran claimed to hack the U.S. drone by exploiting a bug in its

software [121, 92, 80]. Therefore, it remains an open question whether the *Confidentiality*, *Integrity*, and *Availability* (CIA) triad principles, which is a standard model designed to guide policies for information security [57], will be maintained during UAVs software development life-cycle. As a result, verification technique has been applied to avionics embedded software (*e.g.* DSVerifier [37] and HyTech [78]), since the 2000s, to develop systems with high dependability and reliability to ensure both user requirements and system behaviour [46].

Verification is the process of verifying the system accuracy and whether it satisfies the specifications. The standard verification processes are simulation, fuzzing, and model checking [41]. Simulation verification is performed on the system's abstract model, whereas fuzzing is done on the actual system. Simulation and fuzzing involve giving specific inputs and checking whether the outputs are as expected. Model-checking, as the name suggests, is model-based. It is an automatic technique to check the violation of a given (safety) property at a given system depth. It involves developing a simple model which identifies the essential features of the required system. The specifications that are to be tested on the system are usually specified in logical statements. Then a model checker, a software tool, systematically examines all the system scenarios to check whether the system satisfies the specifications [40]. Several research studies were conducted during the past ten years to enable and to solve some issues in UAV systems. These efforts classified into three groups:

- Proposing and developing UAV applications.
- Finding and proposing solutions for UAV technical issues.
- Growing platforms, software architectures, and middleware for UAVs [106, 115, 145].

However, few studies discussed low-level properties verification and even fewer of these studies addressed UAV software vulnerabilities detection, such as concurrency safety, control loop executions and buffer overflow.

1.1 Problem Description

A report published by the United States Air Force's Professional Military Education (PME) listings the causes of failure for UAVs as "insufficient testing before purchase" and suggests that it is common to all failed software [34]. Availability of a proper verification method to test UAV software before the flight would lead to mid-air collision and ground casualty prevention. It would also help prevent any software failure and minimize investments in failed experiments. The recent increase in attack attempts on these unmanned systems has raised concern among defence and commercial manufacturers. With the increasing autonomy level

of these systems, concerns over their use have been ever-growing. These concerns necessitate a cost-effective and safe verification method environment for testing the accuracy of various security implementations in UAV software. UAV software verification's main challenge is how to clarify with a high degree of certainty that the system meets its requirements. Indeed, addressing multiple environment variations, loss of connectivity and contested communication are some of the crucial aspects of such a verification due to the dependency of UAVS security on communication. According to Haojie et al. [60], concurrency bugs are extremely difficult to detect and are one of the most challenging vulnerabilities to verify. This is because the mission planning software of multiple UAV systems involves concurrency as it continually interacts with the environment in which the UAVs operate. Thus, it is impossible to verify UAV software from this bug using traditional testing thoroughly [136]. Also, it is difficult to predict in advance what kind of situations can occur during the missions to conduct testing, which therefore involves additional time and memory processing [58, 128, 67].

Based on those factors, this dissertation aims to seek answers to the following research questions:

- **RQ1**: What are the software vulnerabilities of the system in question? How can these vulnerabilities verify?
- **RQ2**: Are verification results using bounded model checking (BMC) or fuzzing approaches practical for verifying UAV software, and can concurrency bugs and other UAV software vulnerabilities be detected based on previous techniques?

The above outlines the issues involved in addressing the challenge of UAV vulnerability detection. The system developed and described in this work represents an implementation of a solution whereby, in the generation of sufficiently inductive invariants that can guide the BMC engine to successfully and precisely detected UAV vulnerabilities.

1.2 Research Objectives and Contributions

This dissertation primarily focuses on developing and testing a verification framework algorithm called $DepthK$ [124], to verify UAV software security requirements and to detect any potential software vulnerabilities. As a prerequisite, detailed threat modelling and risk evaluation have also performed and discussed in this work. Although more detailed requirements of such a testbed have been discussed in the following Chapters, it should be clear that in terms of verification and detection of the UAV software vulnerabilities such as overflow and concurrency bugs, the process remains computationally expensive. In this work, we demonstrate different verification capabilities such as BMC, Invariant reference,

and fuzzing techniques to verify UAV software's correctness. Therefore, all the discussion from here onwards may refer to fuzzing, BMC and invariant reference. The outcome of this research employs the BMC, Invariant reference, and Fuzzing techniques during design and verification of UAV software, by assisting UAV developers with an efficient approach, which brings more confidence and less effort than traditional simulation tools, mainly the ones that require manual intervention. Therefore, the present study's main contribution is to introduce a verification methodology implemented in the DepthK tool and its theoretical basis, which investigates different types of UAV software vulnerabilities available in various domains.

To sum up, the following technical contributions have been made in this dissertation:

- Detailed threat analysis using the CIA triad to identify UAV software vulnerabilities in order to provide:
  - details to specify requirements and define the design of the proposed verification framework.
  - list of vulnerability-based attacks that need to be tested and understanding of their detailed anatomy for verification implementation purposes;
- Research and improve the existing BMC and fuzzing based approaches;
- Apply the proposed methodology to verify UAV software correctness considering its complex vulnerabilities and confirm that it can solve real-world benchmarks and compete with state-of-the-art software verifiers.

An extensive verification study has conducted to reveal the achievable performance of the proposed framework methodology using a broad range of affecting benchmarks extracted from the international competition on software verification (SV-COMP) 2018[1] and 2019[2]. Most importantly, the impact of the number of vulnerabilities detected and the verification mode has also been investigated. These studies prove the correctness and the achievable performance of the developed framework. For example, the SV-COMP 2019 competition consists of 10,522 verification tasks. The proposed method has a confirmation rate of 98% of the verification tasks for which the verifier answered with True, and 71% answered with False; this method outperforms the existing BMC solution records as 94%, 63%, respectively.

[1]https://sv-comp.sosy-lab.org/2018/results/results-verified/
[2]https://sv-comp.sosy-lab.org/2019/results/results-verified/

1.3 Outline of the Solution

This section outlines the solution framework designed to verify UAV software performance in a rule-based system. This solution divided into two parts to verify UAV software: (1) *the investigation of UAV software and associated threats*; for this purpose, DJI Tello and Parrot bebop drones investigated besides various avionics embedded system benchmarks and (2) *the examination and development of existing verification techniques*. Fuzzing and BMC techniques have been chosen as the most appropriate for the proposed verification approach. It also demonstrates why BMC with invariant reference has been chosen as the most appropriate for designing and implementing the proposed methodology. For more information, see Section 3.3.

1.3.1 Investigation of UAV Software & Associated Threats

During the software development life cycle for UAV systems, developers must take into account all characteristics of the UAV software infrastructure that glues together the UAV hardware, operating system, network stack, and application [106]. Therefore, efforts should concentrate on developing and integrating security functions in the initial phases of software design, hence effectively achieving different security requirements such as confidentiality, integrity, and availability. A simple outline of the threats posed to a UAV system and possible attack paths of these threats is given in Figure 1.1 [88].

When a user requests the execution of a specific command from a UAV (e.g. flight command), UAV software collects the data about this activity and stores it in logs with a .dat or .bin file extension. The critical information of each collected activity is the commands involving the users logging in or controlling functions from a high level, which shows the actual event that took place on the UAV System and its results. On the other hand, real-time tracking is done by connecting to the UAV via Wi-Fi and sending commands for data dumps and status updates logs through the connection. Therefore, a possible method for the verification of a UAV software would be the analysis of the syntax of each command a user applied (*e.g.,* proprietary software) or by verifying open-source software to investigate if the proposed verification methodology would cause an exception and reveal hidden vulnerabilities in its system.

As shown in Figure 1.1, the threats on the three main components of the UAV CIA triad are mostly software-based, such as malware, falsifying control commands, and malicious attack. Threats also include using malicious code on GCS or existing subroutines of the UAV software, such as exploiting a buffer overflow vulnerability that aims to overflow the

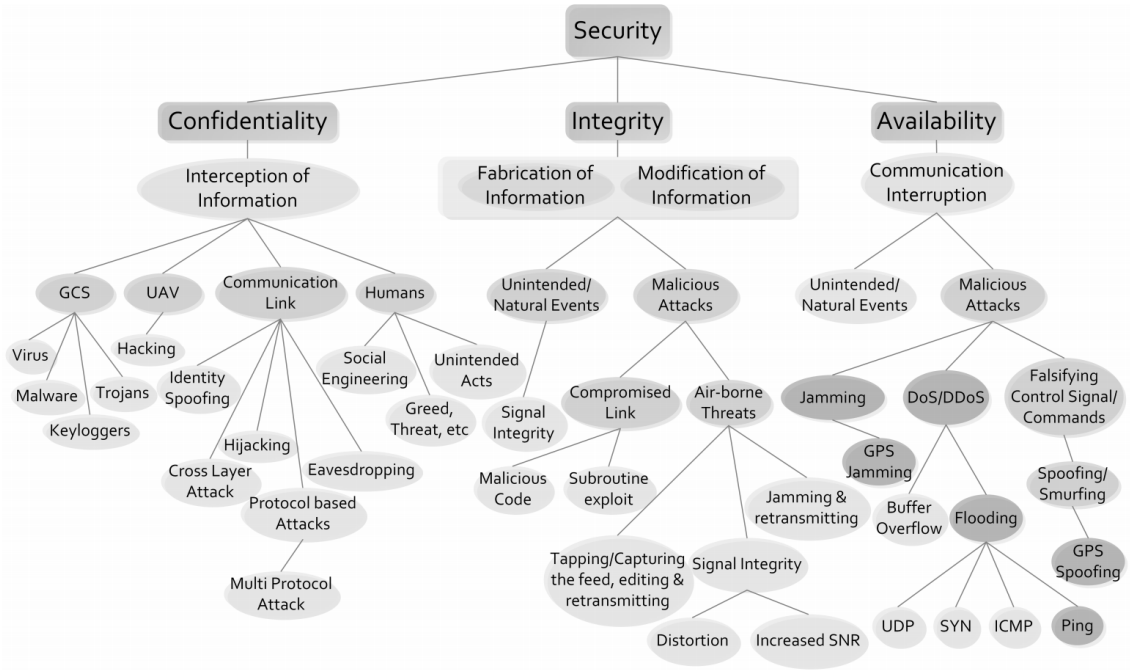

Figure 1.1 Threats investigation using CIA triad

software's buffer memory. The four main ingredients of the UAV model that are vulnerable are the UAV software, GCS, communication systems, and human users. CIA threats were also showing that software-based threats can affect UAV overall security. Hence, this study focuses on UAV software investigations and their associated threats. An approach to implementing this method has been discussed and implemented in more detail in Chapter 3 and Chapter 5.

1.3.2 Examination & Development of Existing Verification Techniques

Software testing and verification techniques are essential for developing systems with high dependability and reliability requirements. Recently, various techniques have emerged to detect software vulnerabilities. Fuzzing and BMC are among the most successful means for software testing and verification techniques for identifying software weaknesses [104]. Therefore, these approaches were investigated further in order to be developed to fit the UAV software complexity structure.

BMC technique limits the visited regions of data structures (e.g. arrays) and the number of related loop iterations. Thus, BMC restricts the state space that needs to be explored during verification so that fundamental errors in applications can be found [39, 103, 43, 86]. However, BMC tools are susceptible to exhaustion of time or memory limits when verifying programs with loop bounds that are too large [22].

On the other hand, the fuzzing technique [112, 104] can create a substantial amount of anomaly data, feed it to a compatible target, and monitor the target behaviour for any potential vulnerabilities. Still, it required a considerable large number of malformed input to maintain high code coverage [112, 3].

Given the research's applied nature, the software verification techniques used fuzzing and BMC were reviewed and assessed. These techniques have a particular bearing on how verification is undertaken within the industry. When reviewing the academic literature, it identified that the existing verification algorithms considering drones software have their limitations [55, 132, 29, 74, 3]. As a result, a new verification methodology developed for verifying UAV software. $DepthK$[3], represents the latest advancements in verification technology and is designed to enable the creation of robust, secure, and interoperable verification software. This method has been discussed in more details in Section 4.3.

1.4 Organization of the Dissertation

The remainder of this work organised as follows.

Chapter 2, describes fundamental concepts about the main verification techniques used for the proposed verification algorithm, along with implementation aspects, theoretical fundamentals, and how the UAVs work.

Chapter 3, the recommended verification methodology for UAV software presented, including a detailed description of each technique and possible attacks. Also, the verification tool implemented in $DepthK$ tool is explained.

Chapter 4, in turn, tackles the performed verification experiments and presents comprehensive performance evaluation results. This chapter also presents extensive evaluation focusing on gauging whether the new verification framework performs decent enough compared with other software verifiers. Tests were performed, and trends were plotted for various safety properties. This chapter also describes how those same outcomes were reproduced and validated.

Chapter 5, describes the anatomy and implementation details of various attacks identified during the threats analysis phase. This chapter also describes the exploitation of some of the UAV software vulnerabilities and how the proposed verification methodology for UAV software introduced. It also includes a detailed explanation of the verification of each safety properties verified by DepthK [4].

Finally, chapter 6 concludes this dissertation by summarizing the research findings and proposes future research topics.

[3]https://github.com/hbgit/depthk

CHAPTER 2

Background

2.1 UAVs

UAV systems are complex distributed systems that share with other distributed systems their heterogeneity, safety, and reliability difficulties in addition to their unique challenges represented by high-safety requirements and other flying related concerns. UAV design includes a communication link between UAV and pilot, ground control station and accessories (*e.g.*, gimbal and payload). A variety of UAV model has been developed; some of these models include Fixed-wing aircraft, chopper, multi-copter, and commercialized UAV. Figure 2.1 illustrates subsystems and modules of the UAV design. Many UAVs are developed to function autonomously and in large networks. With this technology prevalent, cyber-security advancements need to be made for the safe operation of large, autonomous UAV networks.

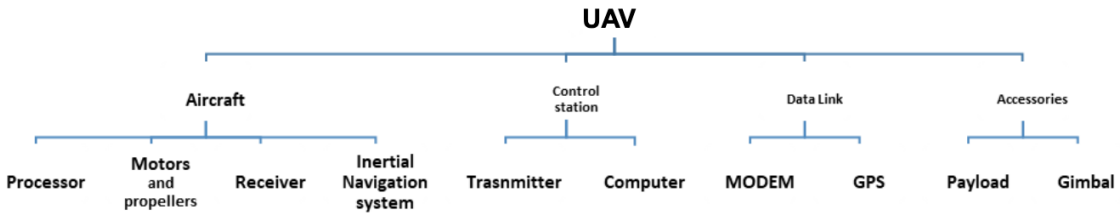

Figure 2.1 UAV design

All of the UAVs are specified for a specific mission and have their zeros and ones. More about UAV systems have been discussed in Section 5.2.1.

2.1.1 UAV Security

UAVs' essential operation, cyberattacks on UAVs, UAV forensics, and secure communication with UAVs are topics of critical importance to UAV security. The most intuitive starting

point is their systems when discussing UAV air operations, including their potential cyber vulnerabilities and other inherent threats. However, before discussing the cyber-security aspects of UAVs, the way UAVs operate must be understood. UAV system is a combination of hardware and software which can accept inputs, process data and, with the processed data, generate information for storage and output[12]. UAVs use Global System for Mobile Communication (GSM) to maintain network connectivity. UAVs also use Unmanned Traffic Management platforms (UTM) that are used to secure low-level operations by pinging the UAV specific commands based on its proximity location to the platform. Whilst the concept's demonstration can become increasingly complex, the basic principles remain constant.

When cheap, consumer-grade drones became available, their flight computers were hacked almost instantly [117]. UAVs' current developments in information security involve encryption methods and new network models, but that barely considers external security threats. As a result, the most advanced UAV and their systems were susceptible to cyber-attack [140, 90, 92, 110], it is asserted that the cyber-vulnerabilities of UAV systems have created a credible threat to air traffic which is severe, widespread and persistent. Therefore, in UAV systems, the need for reliability and autonomy has already motivated other researchers regarding the application of formal methods. However, the present work differs from previous approaches once it investigates the different complicated vulnerabilities in the low-level implementations and the best method to verify each of these vulnerabilities.

2.1.2 UAV Software

UAV software simplifies flight planning, stores sensor data, processing, and analyses UAV data to provide intelligence survey flight plan and platform configuration. When a user requests a command, several system calls are requested by this command from the UAV software. Each of these system calls is responsible for performing an elementary operation required for the command's execution and might be called more than once. UAV software are often employed in different processes (*e.g.*, Transport, Security, surveillance, and rescue operation..etc.). Besides, UAV software will support service providers to optimise UAV input and support their decisions making accordingly. A good UAV software at least must include the automation of UAV flight plans, augmented view, Geo-rectification of images and 2D/3D models generation[134]. Since UAV usage's operational expansion, their vulnerability to cyber-attack has been increasingly highlighted, hence the importance of understanding the proper and efficient ways to secure UAVs greatens. Given this, it is becoming increasingly essential to protect the CIA triad of the information processed by those systems, where software verification can play an important role in relaxing these challenges.

2.1.3 CIA Threat Summary

UAV is a cyber-physical system in which it is vulnerable to most network-oriented cyber-attacks. Still, some of them can be more serious than others and lead to a more vulnerable UAV system state. Therefore, it is essential to prioritize threats according to their risks and their impact on the system once occurred. Based on this priority, threats should be addressed, and proper mitigation measures should take place. Threats analysis to a UAV system and possible risks are shown in Figure 1.1. In addition, threat classifications have been presented in Section 3.2, and the results were published in [3, 4] based on the initial analysis. At this research, a new method presented that improves existing verification techniques (*e.g.*, BMC, Fuzzing, and k-Induction), and can efficiently detect UAV software vulnerabilities.

2.2 Overview of Fuzzing

Fuzzing is excessively utilized as a security method for finding vulnerabilities within different systems by dispatching a sequence of test files containing incorrect data and monitors software irregularity. It has played an essential role in improving software development and testing over several decades. Currently, two types of fuzzing techniques exist- generation-based and mutation-based fuzzing. Fuzzing enable a wide range of verification techniques, including automatic detection of security vulnerabilities, patch generation, and automatic debugging, which have been industrially adopted by large companies, including but not limited to Microsoft (SAGE [68]) and IBM (Apollo [8]).

Fuzzing is a useful method for determining bugs in software and assist in cost-cutting with no need for additional intervention. Fuzzing techniques can create substantial random or semi-random data to discover security vulnerabilities in real-world software. Still, it required a considerable large number of malformed input to maintain high code coverage [3]. Since these inputs are random, their unexpected and improper appearance in a target software is highly feasible. If the target software does not reject these malformed inputs, it will hang or crash during fuzz testing. Critical security flaws most often occur because software inputs are not appropriately checked. Software systems that cannot tolerate fuzzing has surety holes [112]. Fuzzing is a speedy and cost-effective method for locating security vulnerabilities on different platforms.

2.2.1 Types of Fuzzers

Fuzzers are reasonably simple to implement; however, they are unlikely to cover various program states. Two main types of fuzzing techniques were improved to tackle this issue: mutation-based and generation-based approaches.

- **Generation-based fuzzers** Generates fault-injected messages as test cases based on protocol through specifications. Generation-based fuzzers *generate* inputs based on these specifications. The generation-based technique can generate inputs for file transfer protocols, network protocols, and file formats [109].
- **Mutation-based fuzzers** Mutation-based creates test cases by *mutating* or manipulate input data that feed into a target. Generally, mutation-based technique are not aware of the expected input format, or specifications [109].

Due to the nature of UAV software, and since the mutation-based approach requires a higher number of test cases than the generation-based [3], the latter will be used in our proposed method. As shown in Figure 2.2, the process firstly classifies the UAV protocol's specification. Secondly, it identifies the packet vector data sent from the software to the target system. Thirdly, it generates relevant data automatically and executes the new test cases. Finally, monitoring exceptions is essential since creating the crash is not beneficial if the packet that causes the fault cannot be specified.

2.2.2 Fuzzing Process

The fuzzing test presented in this work was launched on different UAV models (*e.g.*, Parrot Bebop and DJI Tello). These UAVs are equipped with autonomous path settings, GPS navigation, an altimeter, an accelerometer, a gyroscope, and a mobile app to control the drone. The fuzzing test involves the exploitation of different vulnerabilities when subjected to fuzzing and flooding attacks. Many UAVs have unrestricted communication systems. Services such as FTP, Telnet, and SSH have all been investigated for vulnerabilities. The model discussed in this work can also be used to diagnose other UAVs on the market to see if they are vulnerable to the same vulnerabilities. The attack not only exposes a way to jam communication with the UAV's GPS but also demonstrates new methodologies to investigate new UAV vulnerabilities. Figure 2.2 below illustrates different fuzzing technique steps.

All fuzzers demands program knowledge, applies that knowledge to generate malformed inputs, select inputs to feed to the software being fuzzed, monitor program behaviours in response to each input, and analyse and export any interesting program states for the user

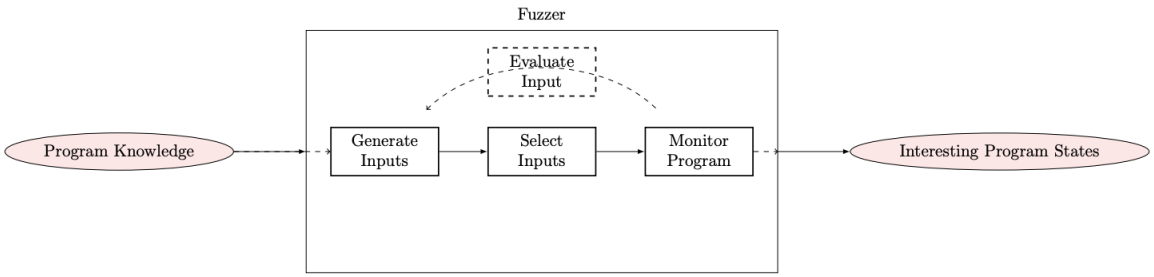

Figure 2.2 Fuzzing Process

to investigate. Fuzzers also can use the results generated while monitoring the program to evaluate input and inform the next input generation stage. However, this process evaluates inputs based on the generating input and its results.

In UAV software, the fuzzing extracts specifications of user actions commands involving the users logging in or controlling functions from a high level. Commands like USER, PASS, QUIT, and REST all deal with either a user logging in, cancelling, or restarting a specific action involving UAV data management. Once the UAV software specifications are extracted, the fuzzing engine is loaded with an extendable fuzzing function list. Initially, the fuzzing engine sets its state to the main functions of the software. It then monitors the input traffic through the GCS (*e.g.*, UAV mobile application), thereby making appropriate transitions and applying fuzzing functions. Any hints the GCS gave of critical services disconnecting were monitored carefully and recorded accordingly. Fuzzing, floods, and DoS attacks have all been used to exploit vulnerabilities in UAV systems. Floods and DoS attacks have been used to crash UAVs, while fuzzing has been used to cause unpredictable UAVs software behaviour. This approach discussed in more detailed in Chapter 5.

2.3 Overview of Model Checking

As the name suggests, model-checking is a model-based technique to verify finite state machine abstraction of the system. It requires developing a simplified model that identifies the systems' essential features, usually in terms of logical statements. The system is represented by finite-state model M and a temporal logic formula ϕ expressing some desired specification. A model checker is used to check M against the specification ϕ. The model checkers outputs will be true if M satisfies ϕ, i.e. $M \models \phi$, or a counterexample if it does not. This model checker implemented in a software tool, systematically examines all the system scenarios to check whether the system satisfies the specifications.

2.4 Bounded Model Checking

BMC techniques, either based on Boolean Satisfiability (SAT) [22] or satisfiability modulo theories (SMT) [13], have been successfully applied to check single and multi-threaded programs and then find subtle vulnerabilities in real programs [39, 103, 43, 20, 35]. The idea behind BMC is to check the negation of a given property at a given depth, *i.e.*, given a transition system M, a property ϕ, and a limit of iterations k, BMC unfolds a given system k times and converts it into a verification condition (VC) ψ, such that ψ is *satisfiable* if and only if ϕ has a counterexample of depth less than or equal to k. Figure 2.3 below depict how BMC works.

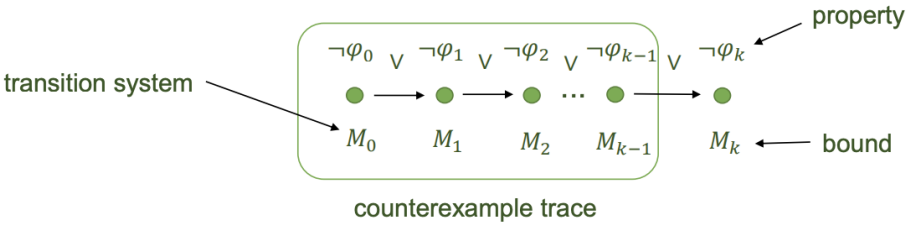

Figure 2.3 Representation of a transition system

BMC based on SAT [22] was initially proposed in the early 2000s to verify hardware designs [22, 23]. Indeed, a group of researchers at Carnegie Mellon University were able to successfully check large digital circuits with approximately 9510 latches, and 9499 inputs, leading to BMC formulae with 4×10^6 variables and 1.2×10^7 clauses to be checked by standard SAT solvers [23]. BMC based on SMT [13], in turn, was initially proposed by Armando *et al.* [7], in order to deal with ever-increasing software-verification complexity.

Generally speaking, BMC techniques aim to check the violation of a given (safety) property at a given system depth. Indeed, given a transition system M, which is derived from the control-flow graph of a program, a property ϕ, which represents program correctness and/or a system's behaviour, and an iteration bound k, which limits loop unrolling, data structures, and context-switches. BMC techniques thus unfold a transition system M k times in order to convert it into a verification condition ψ, which is expressed in propositional logic or a decidable-fragment of first-order logic.

One may notice that BMC analyses only bounded program runs but generates verification conditions that indicate the exact path in which a statement is performed, the context in which a given function is called, and the bit-accurate representation of expressions. In this context, a verification condition is a logical formula (constructed from a bounded program and desired correctness properties) whose validity indicates that a program's behaviour

agrees with its specification [31]. Users can specify correctness properties in our context via *assert* statements or automatically generated from a specification language [11]. If all of a bounded program's verification conditions are valid, then a program complies with its specification up to the given bound. Although BMC was introduced nearly two decades ago, it has only relatively recently been made practical as a result of significant advances in SMT [22]. Nevertheless, the impact of this approach is still limited in practice due to the current *size* (e.g., source code size) and *complexity* (e.g., loops and recursions) of software systems [62].

BMC tools also tend to fail due to memory or time limits if programs with loops whose bounds are too large or cannot be statically determined and verified. In addition, even if a program does not contain a violation up to a given bound k, nothing can be said about $k+1$. Consequently, such limitations have motivated researchers to develop new approaches in order to go deep into a program's search space and, at the same time, prove global correctness. In particular, two possible strategies have been proposed in the literature; in order to achieve that goal: k-induction [55, 132] and IC3 [29, 74], both techniques were explained in Chapter 4 and experimentally evaluated their performance against the proposed verification algorithm.

2.5 Invariant Inference

This section briefly provides background on inference algorithms that motivate our theoretical development in this work. The inference of inductive invariants is a fundamental technique in safety verification and the focus of numerous works [6, 27, 138]. The advance of SAT-based reasoning has led to the advancement of successful algorithms inferring inductive invariants using SAT queries. A prominent example is IC3/PDR [28], which has led to a significant development in the ability to verify hardware systems and extended later to include software systems [26].

In traditional software verification, a human annotates the loops of a given software with invariants, and a decision procedure checks these invariants by proving some verification conditions. This work investigated whether a verification methodology for combining k-induction and loop invariant generation can help automate one of the core problems in verification (discovering appropriate invariants) and explore the goal of pruning state-spaces. Therefore, a verification framework called $DepthK$ described that by giving a procedure for checking invariants and using that checker to produce an invariant inference engine for a given language of possible invariants. As discussed in Chapter 4, this work applied to various classes of invariants; which used to generate different inference procedures that prove safety

properties of the most challenging UAV security (*e.g.*, Concurrency and overflow bugs), and outperforms all previously mentioned techniques (*e.g.*, BMC and IC3).

2.6 Related Work

UAVs and airborne software proliferation has led researchers to propose several integrated development frameworks and develop model-checking applications for UAV software [87]. This section discusses various works that have been done in the area of UAV security and software verification. Since the general UAV system structure initially developed for operating purposes, cyber-security was not a design priority [111]. Therefore, an in-depth analysis of cyber threats and vulnerability identification on these systems still required to identify the systems' correctness and verify whether the system meets the design requirements. Many works focused on improving the UAV system performance, tackling digital filters, or verifying only a single error (*e.g.*, run-time or zero-input errors) using formal verification methods. However, verifying hybrid systems like UAVs is challenging, particularly UAVs software verifications.

In a recent work [130], Rudo *et al.* launched a fuzzing test on a Parrot UAV in order to identify specific patterns and detect vulnerabilities. This approach successfully interrupted GPS navigation, and other potential vulnerabilities were hypothesised. However, fuzzing tests were regarded as a complicated practical method due to UAV systems' structure [37]. Hence, formal verification methods, such as model-checking and deductive verification, became of interest in order to verify the software correctness with high certainty. For instance, NASA employed a model checker tool named *SPIN* in order to verify NASA'S flight missions besides developing formal verification techniques for their satellites intelligent autonomous system [33, 116].

Another work introduced by Wang *et al.* [143], where they proposed a generating algorithm and modelled a simplified flight control system of UAV to automatically check its real-time property and extract real-time property from time-sensitive live sequence charts. Another similar work [93] proposed a UAV-related methodology represented by linear-time temporal logic expressions for building and checking UAV missions specifications with SAT-based techniques to ensure planned areas' reachability. Based on the previous method, another verification technique combined a symbolic execution technique with an abstraction discovered on-the-fly using simulations to falsify hybrid systems software's safety properties [146]. Additionally [136], Sirigineedi *et al.* employed an *SMV* model checker to verify a UAV's behaviour correctness performing a cooperative search mission against the temporal

properties expressed in Computational Tree Logic (CTL). However, vulnerabilities such as concurrency left as future work.

All studies mentioned above have concerned the high-level specs commonly related to flight planning and navigation system. However, this study concentrates on the UAV low-level specifications by examining software implementation issues. Moreover, the above studies can capture some intuition about the underlying system; however, complexity is not covered regarding all involved code parts, mainly related to closed-loop outputs. In this study, our purpose is a proof-by-induction algorithm that combines *k*-induction with invariant inference, which goes deep into a programs' search space to prove global correctness and verify various software safety properties.

Another good work presented an SMT-based BMC verification method implemented in a tool named $DSVerifier$ to verify UAV low-level properties, and the tool was only able to detect overflow and limit-cycle oscillation in digital controllers [37]. This approach also used incremental BMC, *k*-induction and fuzzing techniques. However, their fuzzing approach was unable to identify any violations from UAV attitude control software benchmarks. In addition, the *k*-induction approach was unable to produce inductive invariants to instrument UAV software. On the other side, this work proposed a methodology that successfully applied fuzzing, strengthening inductive invariants, and used *k*-induction with invariants to handle a wide variety of safety properties in order to detect UAV software vulnerabilities.

One may notice that the *k*-induction algorithm has already been implemented and further studied by the software verification community, which led to comparisons between our approach and other similar *k*-induction algorithms. For example, Bradley *et al.* introduced IC3 procedure for safety verification of systems [29, 74] and showed that IC3 could scale on specific benchmarks, where *k*-induction fails to achieve. However, the proposed approach considers various UAV software vulnerabilities, and it also designed an essential contribution towards verifying low-level specifications that not addressed by previous methods [130, 37, 143, 93, 136].

UAV applications present complicated combined-components and impose an ever-increasing level of difficulty in existing verification methodologies [37]. Mainly, software testing has been the traditional method for investigating software vulnerabilities. The comparison between software testing and model-checking techniques [99, 21] has been of interest to some researchers, which showed that the model-checking technique was more competitive in detecting vulnerabilities.

According to Dirk [21], software model checkers are also efficient to be adopted in practice and are also faster and can compete with state-of-the-art software testing tools. The previous studies also recommended a model checking technique for practical applications to

ensure control systems' stability and consider digital controllers' software implementations, which was the case for the proposed methodology. However, difficulties related to low-level implementation effects (*e.g.*, concurrency and overflow bugs) are among the most concerns vulnerabilities threatening UAV software, and only a few studies investigated those problems.

For example, Ismail *et al.* [85] and Bessa *et al.* [14] employed $DSVerifier$ to check overflow vulnerability in UAV software; however, it was not comprehensive and only tested on a few benchmarks. The same algorithms were further developed by Chaves *et al.* [37], which concentrates on UAV attitude control systems and enabling granular limit-cycle oscillation and overflow detection. Another well-known verification tool is $Astree$ [107], which works on preprocessed C code; however, it only verifies overflow and registers dimensioning, meaning it is not compatible with other UAV software concerns.

This study, however, introduced an SMT-based BMC verification method integrated into the $DepthK$ tool and connected different verification techniques to detect various UAV software vulnerabilities [5]. In order to evaluate the proposed approach's effectiveness, the tool applied to real UAVs, embedded-system applications designed for avionics missions and thousands of C benchmarks from SV-COMP competition. The experimental results show that $DepthK$ outperforms all the previous methods when facing the most challenge UAV vulnerabilities, such as concurrency vulnerabilities.

CHAPTER 3

Research Methodology

3.1 Threats and Vulnerability Identification

Research on the current UAV systems, including ground control stations and communications, carried out to verify the data flow in and out of the UAV itself. Besides, attending several UAV security-related meetings within the School of Mechanical and Aerospace. As a result, new methods for detecting UAV software vulnerabilities by manipulating the flowing data were hypothesised. Literature reviews carried out to assess the possibility of the hypothesised method and the best verification methods. High accuracies aircraft models such as Parrot Bebop and DJI drones were used for this purpose. Comparative evaluations conducted for potential UAV vulnerabilities in order to identify the most effective verification method (see Section 4.4.1). Experimental studies also investigated the UAV behaviour following exploiting its vulnerabilities to simulate real cyber-attack scenarios and determine the most effective software threats (see Section 5.4.3). The UAV models, vulnerabilities detection techniques and the verification method will be covered in detail.

3.2 Attack Possibilities

Several cyber-attack possibilities identified after investigating the UAVs system and the possible vulnerabilities of the UAV software. These attacks have categorised into three main elements:

- **Hardware Attack**

  The likelihood of such attacks required the attacker to have physical access to the UAV components directly. An attacker can connect directly to the UAV system and damage it or maliciously reprogram it, allowing unauthorised control of the UAV or unauthorised

access to its data. Hardware attacks can affect the UAV's reliability, compromise control, and compromise the collected data.

- **Communication Attack**

  This attack launched through wireless/GPS communication channels. For example, if an attacker cracks the communication channel encryption, an attacker can gain complete UAV control. Another possibility is an attack that can send false data (*e.g.,* GPS spoofing) through the GPS channels or blind its sensors. The danger of this attack that it can target a UAV from a distance and while in operation. The UAV software also relies on the communication data flows between the communication system and other UAV components, which increases the seriousness of this type of attacks.

- **Software Attack**

  Software vulnerability exploitation will affect the CIA triad. For example, a buffer overflow vulnerability is exploited when an attacker sends carefully crafted input, allowing the execution of arbitrary code within the vulnerable software and initiating unauthorised tasks. Another possible attack by exploiting concurrency vulnerability; UAVs mission required planning software to schedule concurrent tasks because it deals with multiple UAVs on the same mission. UAV concurrency bugs are among the most challenging software vulnerabilities to detect and verify due to their non-deterministic triggers caused by task scheduling and interrupt handling [60]. Some examples of this type of attack include search-attack mission planning, sensory or command, and control data manipulation.

This research focuses on UAV software vulnerability detection and how to verify software correctness. The possibility of exploiting UAV software vulnerabilities such as concurrency and buffer overflow were assumed. However, the study will not discuss the attack techniques; instead, the seriousness of these vulnerabilities and their damage results were considered.

3.3 The Proposed Verification Methodology

3.3.1 Fuzzing

The concept of software fuzzing can be implemented to UAV Guidance, Navigation and Control (GNC) Algorithm. In the UAV system, irregular inputs with expected distributions are not uncommon; however, unexpected, malformed, or completely inconsistent inputs can cause unexpected behaviour and reveal new vulnerabilities. The fuzzing attack can be injected

at several various points in the UAV data flow. For this research, random inputs applied to corrupt the data flow from the controller to the communication system. The hardware used for this testing involved a Parrot Bebop and DJI UAVs, a software-defined radio called bladerf, a raspberry-pi, 2.4 *GHz* antennas, and a laptop with Kali Linux installed. A fuzzing test also introduced in this work is carried through airmon-ng, a tool installed on Kali Linux by default. The bash script used in Fig 3.1, pre-configured with the drone MAC address, called airmon-ng, which ran a series of 1000 packets at ports between 1 to 100. The fuzzing test results were seen, such as the loss of GPS navigation and the drone's control.

```
#!/usr/bin/env bash
#!fuzzing test to corrupt the GPS connection

port=1
while [ "$port" -le 100 ]
do
aireplay-ng -0 1000 -a #Drone Mac Address xx:xx:xx:xx wlan1 #Wifi card
port=$((port+1))
if [ "$port" -eq 100 ]
then
port=1
fi
done
```

Figure 3.1 Bash script used to fuzz UAVs

The same attack was also used to simulate an actual attack in an organised competition sponsored by the British multinational defence, security, and aerospace company (BAE) [4]. These attacks involve a fuzzing test on UAVs affects the flight's stability compared to the typical case. Attack results also include showing little to no video obscuration, block the drone's GPS from displaying its location, and the UAV would be unable to locate or self-navigate. This also indicates that other services can be further exploited by fuzzing specific commands. Therefore, intentional fuzzing tests could be implemented on the UAV system to determine the UAV overall robustness. However, UAV software vulnerabilities, including concurrency and buffer overflow still left missing. This approach is computationally costly, whilst concurrency vulnerability leads to a combinatorial explosion in possible states [83]. For more information, see Section 5.3.

3.3.2 Induction-based Verification

The Inductive approach scales up to the verification of real-world software thanks to its extensibility. One approach to achieve completeness in BMC verification technique is to prove that an invariant (assertion) is k-inductive using SAT/SMT solvers. The main challenge

regarding such a technique relies on computing and strengthening inductive invariants from programs to prove software correctness. In the present work, a verification methodology for combining *k*-induction and loop invariant generation implemented in a tool named $DepthK$ to automatically verify software vulnerability such as concurrency and buffer overflow bugs, in software employed in UAVs, as shown in Fig 3.2.

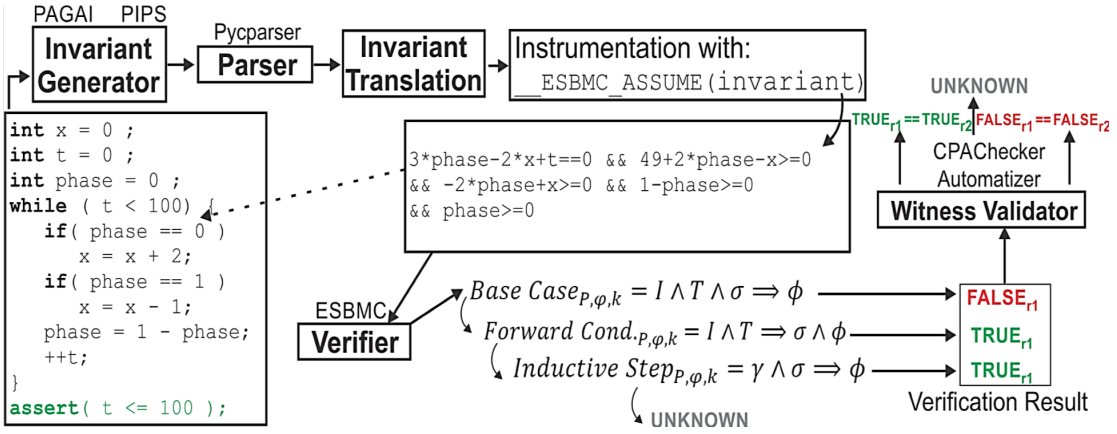

Figure 3.2 The proposed methodology for UAV software verification

$DepthK$ described as a new technique that generates invariants (using external tools), creates a copy of an input program modified by the inductive step of the *k*-induction algorithm to over-approximate its loops. The proposed methodology takes three inputs: the original program P, the property ϕ to be checked, and the chosen invariant tool, either $PIPS$ or $PAGAI$. It returns $FALSE$ if a property violation is found, $TRUE$ if it can prove correctness or $UNKNOWN$. Witness checkers [17] also integrated into $DepthK$ to benefit from guidance information provided after achieving a conclusive result (*true* or *false*), and to provide reliable results.

As described in Fig 3.2, the algorithm tries to find a counterexample up to k steps in the extended algorithm's first step. If no property violation is found, then the forward condition checks whether the completeness threshold is reached at the current k step using unwinding assertion. Finally, the inductive step checks whether the property ϕ holds indefinitely. If ϕ is valid for k iterations, it must be valid for the next ones. This extension aims to convert the *k*-induction algorithm into a bidirectional search approach using the base case as the forward part and the inductive step as the backward one. For more information, see Section 4.3.

The newly proposed method applied to verify C benchmarks from various embedded system applications, and it also verified thousands of benchmarks correctness whilst participating in the Software Verification competition (SV-Comp 2019). The tested benchmarks and the vulnerabilities exhibited are of critical interest to ensure security in UAVs. For more details of the experimental setup and its result, see Section 4.4.1 and Section 4.4.2.

CHAPTER 4

Verification and Refutation of C Programs based on *k*-Induction and Invariant Inference

Omar M. Alhawi, Herbert Rocha, Mikhail R. Gadelha, Lucas Cordeiro and Eddie Batista

Published: International Journal on Software Tools for Technology Transfer (STTT) - 2020

Statement of Contribution of Joint Authorship [1]

Omar Alhawi **(Principal Author)**
Carried out the new development of DepthK approach and participate in the Software Verification competition (SV-Comp 2019), writing and compilation of manuscript, experimental evaluation, interpretation of results, and preparation of tables.
Herbert Rocha **(Research Colleague)**
Preparation of figures and co-author of a manuscript.
Mikhail Gadelha **(Research Colleague)**
Assisted with manuscript compilation, carried out the implementation of the ESBMC tool to support specific functions of DepthK approach, editing and co-author of manuscript.
Lucas Cordeiro **(Principal Supervisor)**
Supervised and assisted with manuscript compilation, provided the background in software verification, interpretation of the verification algorithms, and he also drafted the manuscript.
Eddie Batista **(Research Colleague)**
Editing and co-author of a manuscript

This Chapter is an exact copy of the journal paper referred to above

[1] All authors read and approved the final manuscript.

Abstract

DepthK is a source-to-source transformation tool that employs bounded model checking (BMC) to verify and falsify safety properties in single- and multi-threaded C programs, without manual annotation of loop invariants. Here, we describe and evaluate a proof-by-induction algorithm that combines *k*-induction with invariant inference to prove and refute safety properties. We apply two invariant generators to produce program invariants and feed these into a *k*-induction-based verification algorithm implemented in DepthK, which uses the efficient SMT-based context-bounded model checker (ESBMC) as sequential verification back-end. A set of C benchmarks from the international competition on software verification (SV-COMP) and embedded-system applications extracted from the available literature are used to evaluate the effectiveness of the proposed approach. Experimental results show that *k*-induction with invariants can handle a wide variety of safety properties, in typical programs with loops and embedded software applications from the telecommunications, control systems, and medical domains. The results of our comparative evaluation extend the knowledge about approaches that rely on both BMC and *k*-induction for software verification, in the following ways. (1) The proposed method outperforms the existing implementations that use *k*-induction with an interval-invariant generator (*e.g.*, 2LS and ESBMC), in the category ConcurrencySafety, and overcame, in others categories, such as SoftwareSystems, other software verifiers that use plain BMC (*e.g.*, CBMC). Also, (2) it is more precise than other verifiers based on the property-directed reachability (PDR) algorithm (*i.e.*, SeaHorn, Vvt and CPAchecker-CTIGAR). This way, our methodology demonstrated improvement over existing BMC and *k*-induction-based approaches.

keywords Software engineering, Formal methods, Bounded model checking, *k*-Induction, and Invariant inference.

4.1 Introduction

Computer-based systems have been applied to different domains (*e.g.*, industrial, military, education, and wearable), which generally demand high-quality software, in order to meet a target system's requirements. In particular, (critical) embedded systems, such as those in the avionics and medical domains, impose several restrictions (*e.g.*, response time and data accuracy) that must be met and measured, according to users' requirements; otherwise, failures may lead to catastrophic situations. As a result, software testing and verification techniques are essential ingredients for developing systems with high dependability and reliability requirements, where it is necessary to ensure both user requirements and system behaviour.

Bounded model checking (BMC) techniques, either based on boolean satisfiability (SAT) [22] or satisfiability modulo theories (SMT) [13], have been successfully applied to check single and multi-threaded programs and then find subtle bugs in real programs [39, 103, 43, 20, 35]. The idea behind BMC is to check the negation of a given property at a given depth, *i.e.*, given a transition system M, a property ϕ, and a limit of iterations k, BMC unfolds a given system k times and converts it into a verification condition (VC) ψ, such that ψ is *satisfiable* if and only if ϕ has a counterexample of depth less than or equal to k.

Typically, BMC techniques can falsify properties up to a given depth k; however, they are not able to prove system correctness, unless an upper bound of k is known, *i.e.*, a bound that unwinds all loops and recursive functions to their maximum possible depths. In summary, BMC techniques limit the visited regions of data structures (*e.g.*, arrays) and the number of related loop iterations. Thus, BMC restricts the state space that needs to be explored during verification, in such a way that real errors in applications can be found [39, 103, 43, 86]. Nonetheless, BMC tools are susceptible to exhaustion of time or memory limits, when verifying programs with loop bounds that are too large.

```
unsigned int N=*;
unsigned int i = 0;
long double x=2;
while( i < N ){
  x = ((2*x) - 1);
  ++i;
}
assert( i == N );
assert(x>0);
```

Figure 4.1 Simple unbounded loop program.

For instance, in the simple program illustrated in Fig. 4.1, where the star notation indicates non-determinism, the loop in line 4 runs an unknown number of times, depending on the initial non-deterministic value assigned to `N`, in line 1, and the assertions in lines 8 and 9 hold independently of the `N`'s initial value.

Unfortunately, BMC tools like the C bounded model checker (CBMC) [39], the low-level bounded model checker (LLBMC) [103], and the efficient SMT-based context-bounded model checker (ESBMC) [43] typically fail to verify this family of programs. That happens because such tools must insert an *unwinding assertion* (the negated loop condition) at the end of the loop, as illustrated in Fig. 4.2, line 9, which will fail if k is not set to the maximum value supported by the *unsigned int* data type.

```
unsigned int N=*;
unsigned int i = 0;
long double x=2;
if( i < N ){          }
  x = ((2*x) - 1);    } k copies
  ++i;
}
...
assert( !(i < N) );
assert( i == N );
assert(x>0);
```

Figure 4.2 Finite unwinding done by BMC.

In mathematics, one usually tackles such unbounded problems using *proof by induction*. As a consequence, an approach called *k*-induction has been successfully combined with continuously-refined invariants [20]. There were also attempts to prove, via *k*-induction, that (restricted) C programs do not contain data races [51, 50] or that time constraints are respected [55]. Additionally, *k*-induction is a well-established technique in hardware verification, due to monolithic transition relations present in hardware designs [55, 70, 132]. Finally, regarding the unknown-depth problem mentioned earlier, BMC tools can still be used to prove correctness in those cases, if used as part of *k*-induction algorithms.

In this respect, some approaches usually require invariants to be manually annotated with their values. For example, Donaldson *et al.* [52, 53], were able to increase the precision of the invariant by manually applying *trace partitioning*[122], a refinement technique for abstract domains that enables inference of disjunctive invariants using non-disjunctive domains, while others resort to static analyses for generating invariants [20, 52] that are later refined, which then strengthen associated induction hypotheses. Nonetheless, a complete automatic methodology for producing strong (inductive) invariants would be beneficial, mainly if

consists in direct logical evolution, given that no user interaction or additional refinement is required.

The last paragraphs inspired this work, whose main contribution is a methodology for combining invariant generation and *k*-induction in order to prove correctness of programs written in the C language. Besides, verification-process automation is also tackled, where users do not need to provide loop invariants, *i.e.*, conditions that hold before a loop, are preserved through each loop iteration, and act as properties that are true at a particular program location. Moreover, the adopted invariant-generation tools were integrated through specific interfacing layers, due to their distinct formats, in order to provide a unified solution based on the proposed approach.

As a consequence, we have added a new module to the ESBMC tool, which employs mathematical induction with invariant inference, in order to prove the correctness of programs containing loops and then evaluate the proposed methodology. Such a module implements an algorithm that executes three steps: base case, forward condition, and inductive step [63]. In the first, the goal is to find a counterexample of size *k*, while the second one checks whether loops have been fully unrolled, which is achieved by verifying that no *unwinding assertion* fails, and, finally, the third one verifies if a property holds indefinitely, where the mentioned integration with invariants occurs.

The proposed method infers program invariants to prune the state space exploration and to strengthen induction hypotheses. Additionally, to provide a practical and complete implementation of the proposed methodology, two invariant-generation tools were used: paralléliseur interprocédural de programmes scientifiques (PIPS) [113] and path analysis for invariant generation by abstract interpretation (PAGAI) [76]. The proposed method was implemented in a tool called DepthK [123, 126], which rewrites programs using invariants generated by PIPS or PAGAI, *i.e.*, it adds those elements as assumptions and uses ESBMC to verify the resulting program with *k*-induction [63].

This study is a revised and extended version of previous work [123, 126] and focuses on contributions regarding combination of *k*-induction with invariant inference. In particular, the main original contributions of this paper are as follows:

- We describe the original *k*-induction and our extended version, which combines programs with invariants generated by either PIPS or PAGAI, in order to strengthen its inductive step. In particular, we presented technical details of the combination process for both PIPS and PAGAI. Then, we concluded by presenting an illustrative example in order to demonstrate the effectiveness of our tool. Indeed, the mentioned example was extracted from the International Competition on Software Verification (SV-COMP)

2019 [17], and our extended *k*-induction was able to prove its correctness, but plain *k*-induction was not (Section 4.3).

- We analyze and compare the results of our tool against other existing software verifiers that implement *k*-induction and property-directed reachability (PDR). In particular, we use ESBMC with *k*-induction and interval-invariant generator [43], CBMC [95], 2LS [32], SeaHorn [73], Vvt [72], and CPAchecker-CTIGAR [24] (see Section 4.4). Experimental results showed that *k*-induction with invariant inference could handle a wide variety of safety properties, in typical programs with loops and embedded software applications from the telecommunications, control systems, and medical domains. Our *k*-induction proof rule with polyhedral program invariants was able to solve 2223 verification tasks, *i.e.*, it has proved correctness in 602 and found property violations in 1621 benchmarks. These results outperformed other software verifiers, including 2LS, CBMC and ESBMC, in some particular categories (*e.g.*, *SoftwareSystems* and *ConcurrencySafety*), which thus demonstrates improvement over existing BMC and *k*-induction-based approaches. Besides, our proposed method using PAGAI confirms the hypothesis that DepthK is competitive when compared to the best available PDR-based tool implementations.

Outline. In Section 4.2.1, we first give a brief introduction to the BMC and *k*-induction techniques and also compare them with PDR (or incremental construction of inductive clauses for indubitable correctness - IC3) [29, 74]. Section 4.3 presents our induction-based verification algorithm using polyhedral invariant inference for specifying pre- and post-conditions, which works for C programs. In Section 4.4, the results of our experiments are described by using several software-model-checking benchmarks extracted from SV-COMP and embedded systems applications. In Section 4.5, we discuss the related work and, finally, Section 4.6 presents this work's conclusions.

4.2 Background

4.2.1 Bounded Model Checking

BMC based on SAT [22] was initially proposed in the early 2000s to verify hardware designs [22, 23]. Indeed, a group of researchers at Carnegie Mellon University were able to successfully check large digital circuits with approximately 9510 latches, and 9499 inputs, leading to BMC formulae with 4×10^6 variables and 1.2×10^7 clauses to be checked by

standard SAT solvers [23]. BMC based on SMT [13], in turn, was initially proposed by Armando *et al.* [7], in order to deal with ever-increasing software-verification complexity.

Generally speaking, BMC techniques aim to check the violation of a given (safety) property at a given system depth. Indeed, given a transition system M, which is derived from the control-flow graph of a program, a property ϕ, which represents program correctness and/or a system's behaviour, and an iteration bound k, which limits loop unrolling, data structures, and context-switches. BMC techniques thus unfold a transition system M k times, in order to convert it into a verification condition ψ, which is expressed in propositional logic or in a decidable-fragment of first-order logic. For example, ψ is *satisfiable* if and only if ϕ has a counterexample of depth less than or equal to k. The propositional problem associated with SAT-based BMC is formulated as [23]

$$\psi_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \neg\phi_k, \tag{4.1}$$

where ϕ_k represents a safety property ϕ at step k, I is the set of initial states of M, and $R(s_i, s_{i+1})$ is the transition relation of M at time steps i and $i+1$. Hence, the equation $\bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$ means the set of all executions of M with length k and $\neg\phi_k$ represents the condition that ϕ is violated in state k, which is reached by a bounded execution of M with length k. Finally, the resulting (bit-vector) equation is translated into conjunctive normal form in linear time and passed to a SAT solver for checking satisfiability. Eq. (4.1) can be used to check safety properties [118] (e.g., deadlock freedom), while liveness ones (*e.g.*, starvation freedom) that contain the linear-time temporal Logic (LTL) operator F are verified by encoding $\neg\phi_k$ in a loop within a bounded execution of length at most k, such that ϕ is violated on each state in that loop [108]. This way, Eq. 4.1 can be rewritten as

$$\psi_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \left(\bigvee_{i=0}^{k} \neg\phi_i \right), \tag{4.2}$$

where ϕ_i is the propositional variable ϕ at time step i. Thus, this formula can be satisfied if and only if, for some i ($i \leq k$), there exists a reachable state at time step i in which ϕ is violated.

One may notice that BMC analyzes only bounded program runs, but generates verification conditions (VCs) that reflect the exact path in which a statement is executed, the context in which a given function is called, and the bit-accurate representation of expressions. In this context, a verification condition is a logical formula (constructed from a bounded program and desired correctness properties) whose validity implies that a program's behaviour agrees with its specification [31]. Users can specify correctness properties in our context via *assert*

statements or automatically generated from a specification language [11]. If all of a bounded program's VCs are valid, then a program complies with its specification up, to the given bound.

BMC tools tend to fail due to memory or time limits if programs with loops whose bounds are too large or cannot be statically determined are verified. In addition, even if a program does not contain a violation up to a given bound k, nothing can be said about $k+1$. Consequently, such limitations has motivated researchers to develop new verification techniques, in order to go deep into a program's search space and, at the same time, prove global correctness. In particular, two possible strategies have been proposed in the literature, in order to achieve that goal: k-induction [55, 132] and IC3 [29, 74], which are briefly described in the following sections.

4.2.2 Induction-based Verification of C Programs

One approach to achieve completeness in BMC techniques is to prove that an invariant (assertion) is k-inductive using SAT/SMT solvers [55, 132]. The main challenge regarding such a technique relies on computing and strengthening inductive invariants from programs, in order to prove global correctness.

In particular, full verification requires, as a crucial step, inference of each loop with a **loop invariant** [61], which is a logical formula that is an abstract specification of a loop. Therefore, loop invariants provide the means to reason about loops and to prove their correctness. According to the Xujie *et al.* [133] inferring loop invariants enables a broad and deep range of correctness and security properties to be proven automatically by a variety of program verification tools spanning type checkers, static analyzers, and theorem provers. Moreover, loop invariants must be inductive in order to check satisfiability for the corresponding VCs, as described by Bradley and Manna [31].

Si *et al.* [133] define loop invariant inference by introducing Hoare logic [79] for proving program-correctness assertions. Let P (pre-condition) and Q (post-condition) denote predicates over program variables and also let S denote a program under evaluation. Based on Hoare rules, such triples can be inductively derived over the structure of S. This way, we can highlight the following one regarding loops:

$$\frac{P \implies I \quad \{I \wedge B\}S\{I\} \quad (I \wedge \neg B) \implies Q}{\{P\} \quad \textbf{while } B \textbf{ do } S \quad \{Q\}} \tag{4.3}$$

where predicate I (the inductive invariant) is called a loop invariant, *i.e.*, an assertion that holds before and after each iteration, as shown in the premise of the rule, and B is a predicate

on a program state. Thus, if a loop is equipped with an invariant, proving its correctness means establishing the two following hypotheses [61]:

- The initialization ensures the invariant, which is called initiation property;
- The body preserves the invariant, which is called consecution (or inductiveness) property.

For instance, consider the C-code fragment shown in Figure 4.2. Suppose that one wants to prove that $P : x > 0$ is invariant. In order to attempt proving the invariant property P, one can apply induction considering that the underlying software-model checker supports IEEE floating-point standard (IEEE 754) [84, 69]:

- In the base case, it holds initially because

$$\underbrace{N = * \wedge i = 0 \wedge x = 2}_{\text{initial condition}} \implies \underbrace{x > 0}_{\text{P}};$$

- In the inductive step, whenever P holds for k loop unwindings, it also holds for $k+1$ steps, *i.e.*,

$$\underbrace{x > 0}_{\text{P}} \wedge \underbrace{x' = 2 * x - 1 \wedge i' = i + 1}_{\text{transition relation}} \implies \underbrace{x' > 0}_{\text{P'}}.$$

Specifically, if we consider the IEEE 754 standard [84, 69], then the invariant $x > 0$ holds initially and after each iteration and x tends to infinity after 128 iterations, so $x > 0$ is a candidate for a loop invariant. Nonetheless, this invariant is not inductive, given that $x > 0$ before an initial iteration does not ensure that $x > 0$ after each iteration, given that if we initially assign $x = 0.9$, then $x < 0$ after the fourth iteration. As a consequence, even if invariant-generation procedures successfully compute such assertions, which are indeed invariant, those must be inductive, so that k-induction verifiers can automatically prove global correctness. In this specific example, an inductive invariant would be $x > 1$, given that if $x > 1$ holds before the initial iteration, then $x > 1$ also holds after k iterations.

Several invariant-generation algorithms discover linear and polynomial relations among integer and real variables, in order to provide loop invariants and also to discover memory "shapes", in programming languages with pointers, such as those used in PIPs and PAGAI [113, 76]. The current literature regarding that also reports a significant increase in effectiveness and efficiency, while outperforms all previous implementations of k-induction-based verification algorithms for C programs, using invariant generation and strengthening, mostly based on interval analysis [20].

Novel verification algorithms for proving correctness of (a large set of) C programs, by mathematical induction and in a completely automatic way (*i.e.*, users do not need to provide loop invariants), were recently proposed [63, 20, 32, 123, 50, 126]. Additionally, k-induction based verification was also applied to ensure that (restricted) C programs (1) do not contain violations related to data races [51], considering the Cell BE processor, and (2) do respect time constraints, which are specified during system design phases [55]. Apart from that, k-induction is easily applied, due to the monolithic transition relation present in such designs [55, 132, 70].

Note that k-induction with invariants has the potential to be directly integrated into existing BMC approaches, given that the induction algorithm itself can be seen as an extension after k unwindings. It is possible to generate program invariants with other software modules, which are then translated and instrumented into an input program [123].

4.2.3 Property-directed Reachability (or IC3)

While BMC is very effective in finding counterexamples, it is indeed incomplete, due to the bound limitation. This weakness motivated the development of IC3 and other complete SAT-based approaches. In particular, Bradley *et al.* [29, 74] introduced IC3, which is also known as PDR. IC3 aims to find an inductive invariant F stronger than P, *i.e.*,

$$\begin{aligned} &INIT \Rightarrow F \\ &F \wedge T \Rightarrow F' \\ &F \Rightarrow P, \end{aligned} \tag{4.4}$$

where $INIT$ describes the set of initial states and T represents the set of transitions, by learning relatively inductive facts (incrementally) locally. Indeed, that is carried out by iteratively computing an over-approximated reachability sequence $F_0, F_1, ..., F_{l+1}$, such that

$$\begin{aligned} &F_0 = INIT \\ &F_i => F_{i+1} \\ &F_i \wedge T \Rightarrow F'_{i+1} \\ &F_i \Rightarrow P. \end{aligned} \tag{4.5}$$

In summary, starting from the initial states, every assignment that satisfies the current clause F_i also satisfies the next one (F_{i+1}), every reachable state satisfies the next clause, and a given

property is satisfied in every clause, *i.e.*, P is an invariant up to $k+1$. As a result, if Eq. (4.5) is performed, F becomes an inductive invariant stronger than P, as shown in Eq. (4.4).

In fact, IC3 is a procedure for safety verification of systems and some studies have shown that IC3 can scale on specific benchmarks, where k-induction fails to succeed. In particular, the success of IC3 over k-induction procedures is due to the ability of the former to guide a search for inductive instances with counterexamples to inductiveness (CTIs) of a given property. Besides, the previous SAT-based approaches require unrolling the transition relation T (cf. Eqs. 4.4 and 4.5), in order to search for an inductive invariant and to strengthen it; however, IC3 performs no unrolling, given that it learns relatively inductive facts locally.

A CTI is a state (more generally, a set of states represented by a cube, *i.e.*, a conjunction of literals) that is a counterexample to consecution [30]. Consider again the C-code fragment shown in Figure 4.2, where $P : x > 0$ is an invariant, but assume now that x has been initialized with a non-deterministic value, in order to make it harder to infer an invariant, as given by

$$\underbrace{x > 0}_{\text{P}} \wedge \underbrace{x' = 2 * x - 1 \wedge i' = i + 1}_{\text{transition relation}} \not\Longrightarrow \underbrace{x' > 0}_{\text{P'}}.$$

In that specific example, one possible CTI returned by a SAT/SMT solver is $x = 0$. If this particular state is not eliminated from the search space performed by the solver, then the invariant P cannot be established, since it is not inductive, given the initial assignment to x. Indeed, the generated inductive assertion should establish that the CTI $x = 0$ is unreachable and if such an inductive assertion does not exist, then other CTIs can be examined instead (*e.g.*, 0.1, 0.2, ..., 0.9). As a consequence, the resulting lemmas must be strong enough that consecutively revisiting a finite set of CTIs will eventually lead to an assertion, which is inductive relative to them, thus eliminating the proposed CTI. In the mentioned example, the loop invariant candidate $x > 1$ is inductive and thus eliminates all possible CTIs in our running example.

Recent work has been done to improve the IC3's strengths further, in order to prove safety properties. One notable study was performed by Jovanović *et al.* [91], which presents a reformulation of the IC3 technique by separating reachability checking from inductive reasoning. In particular, those authors further replace the regular induction algorithm by the k-induction proof rule and show that it provides more concise invariants than the original approach proposed by Bradley [29]. Additionally, the mentioned authors implemented that proof rule in the SALLY model checker[2], using the SMT solver Yices2[3], in order to perform

[2]https://github.com/SRI-CSL/sally
[3]http://yices.csl.sri.com/

the forward search, and MathSAT5[4], to perform backward search. Finally, they showed that their proposed algorithm could solve several real-world benchmarks, at least as fast as other existing approaches.

4.3 Induction-based Verification of C Programs Using Invariants

In this section, we describe the main contribution of the present paper: a verification methodology for combining *k*-induction and loop invariant generation, which was implemented in a tool named as DepthK[5]. The first step of our methodology consists in generating invariants for a given ANSI-C program, which is performed with external tools to strengthen the associated inductive step. Indeed, PIPS and PAGAI were used for such a purpose, with invariants included as comments in different formats, which led to the development of distinct integration layers for each invariant generator, as described in Sections 4.3.3 and 4.3.4. In that sense, future invariant-generation tools would then only need new integration layers, when used along with our verification methodology. Moreover, one may notice that PIPS and PAGAI are suitable for the C language and have the potential to handle a wide variety of safety properties [101].

Although other invariant generators beyond PIPs and PAGAI do exist, such as accelerated symbolic polyhedral invariant computation (ASPIC) and integer set library (ISL), the latter do not handle automatic abstraction and some specific details of the C programming language, such as pointer arithmetic, as mentioned by Maisonneuve *et al.* [101]. Moreover, PIPS and PAGAI present different configuration options, which lead to different results and can also be explored with the goal of pruning state-spaces. Specifically, PIPS [101] is an inter-procedural source-to-source compiler framework for C and Fortran, based on automatic static analysis, which relies on polyhedral abstraction of program behaviour for inferring invariants, while PAGAI [76] is a source code analysis algorithm based on abstract interpretation with linear domains (products of intervals, octagons, and polyhedra) and path focusing, which can generate inductive invariants.

4.3.1 The *k*-Induction Algorithm

Algorithm 1 shows an overview of the *k*-induction algorithm used in ESBMC [63], which is an extended version of the original *k*-induction initially proposed by Eén and Sörensson [55].

[4]http://mathsat.fbk.eu/
[5]https://github.com/hbgit/depthk

It takes a program P, as input, and returns FALSE if a property violation is found, TRUE if it can prove correctness, or UNKNOWN.

Algorithm 1 The k-induction algorithm.

Input: Program P
Output: TRUE, FALSE, or UNKNOWN

```
1  begin
2      k = 1  while k <= max_iterations do
3          if baseCase (P, k) then
4              show the counterexample s[0...k]  return FALSE
5          else if forwardCondition (P, k) then
6              return TRUE
7          else
8              k=k+1  if inductiveStep (P, k) then
9                  return TRUE
10             end
11         end
12     end
13     return UNKNOWN
14 end
```

In the base case (lines 4-6), the algorithm tries to find a counterexample up to k steps. If no property violation is found, then the forward condition (lines 7-8) checks whether the completeness threshold[6] is reached at the current k step. Finally, the inductive step (lines 11-12) checks whether the property ϕ holds indefinitely. If ϕ is valid for k iterations, then it must be valid for the next ones. The algorithm runs up to a certain number of repetitions and only increases the value of k if it cannot falsify the property during the base case. One may notice that k is incremented only at the start of the else branch, on line 10. In our benchmarks, we also noticed that computational resources are wasted if we start with $k = 1$ in the inductive step since loops are usually unfolded at least two times. The properties checked by each step are generated during their execution, as follows. In the base case and also in the inductive step, in addition to user-defined properties (using `assert` statements), safety properties such as out-of-bounds checks, pointer validity, and division by zero are derived from a program and checked. In the forward condition, only the completeness threshold is checked, which is done in a C program, by using *unwinding assertions*, *i.e.*, it verifies whether all loops were unrolled entirely. There exists no need to check any other properties in the forward condition, as all of them were already checked for the current step k. Those properties are used to assign a non-deterministic value of program variables that participate in a loop.

[6]The completeness threshold defines a bound k such that, if no counterexample of length k or less to a given LTL formula is found, then the formula, in fact, holds over all infinite paths in the model [94]

Although *k*-induction is a successful technique to falsify or prove correctness, the over-approximation employed by the inductive step is unconstrained and might present spurious counterexamples. For example, traces produced by a failed inductive step in *k*-induction are a feasible sequence of *k*+1 transitions from an arbitrary loop iteration to an error state, where that loop iteration itself may be unreachable. Currently, our main idea as described by Gadelha *et al.* [64], is to search simultaneously forward, from the initial state, and backward, from the error state, whose procedure stops if those two searches meet halfway. This extension aims to convert the *k*-induction algorithm into a bidirectional search approach by using the base case as the forward part and the inductive step as the backward one.

4.3.2 Extended *k*-Induction Algorithm

Our technique generates invariants (using external tools). It creates a copy of an input program, which is modified by the inductive step of the *k*-induction algorithm in order to over-approximate its loops. Algorithm 2 describes our extended *k*-induction algorithm combined with invariants.

Algorithm 2 Our extended *k*-induction algorithm.

Input: Program P, Tool T
Output: TRUE, FALSE, or UNKNOWN

```
15 begin
16   Inv = genInvariants (P, T)  P' = combine (P, Inv)  k = 1  while k <= max_iterations
       do
17     if baseCase (P', k) then
18       show the counterexample s[0...k]  return FALSE
19     else if forwardCondition (P', k) then
20       return TRUE
21     else
22       k=k+1  if inductiveStep (P', k) then
23         return TRUE
24       end
25     end
26   end
27   return UNKNOWN
28 end
```

Similarly to the original *k*-induction algorithm, our extended version returns FALSE, TRUE, or UNKNOWN, depending on the result of each step. However, it takes three inputs: the original program P, the property ϕ to be checked, and the chosen tool, which is either PIPS or PAGAI.

In the first step of the extended Algorithm 2 (line 16), the chosen tool T is called and it tries to generate invariants for program P. In case of tool failure or no invariant generated, it returns $\varnothing$; otherwise, it returns a set of invariants Inv. The second step is to combine the set of invariants Inv and the original program P, in order to create P' (Line 16). This process is specific to each tool, as invariants are generated in different formats, and, as a consequence, such elements need to be preprocessed before being combined with P. Function `combine` is defined in

$$\texttt{combine} := \left[\; P' := ite(Inv = \varnothing, P, P \wedge Inv) \; \right], \tag{4.6}$$

where the new program P' is the result of an ite operation. If the set of invariants is $\varnothing$, P' is the original program P; otherwise, $P' = P \wedge Inv$. The technical description of the combination of the original program with invariants generated by PIPS and PAGAI is described in Sections 4.3.3 and 4.3.4, respectively.

Finally, the last modification to the base algorithm is to use program P' in every step, during verification, instead of P (lines 17, 19, and 22).

In order to provide reliable results, we have integrated the available witness checkers [17] into DepthK (*i.e.*, CPAchecker and Ultimate Automatizer). After achieving a conclusive result (*true* or *false*), DepthK generates an exstensible markup language (XML) file that contains all program states, *i.e.*, from the initial state to the bad one, which lead to a given property violation. This file is known as the *witness file*. Currently, there exist two state-of-the-art tools able to perform witness validation: CPAchecker [15] and Ultimate Automizer [75]. DepthK currently handles this witness file as follows:

1. It is submitted to CPAchecker [15] for validation and, if the DepthK's result is confirmed, then it is provided to users;

2. If CPAchecker is unable to confirm the result provided by DepthK or if there exists an internal failure in CPAchecker, then it is submitted to Ultimate Automizer [75] and, if the DepthK's result is confirmed, that is provided to users;

3. If both tools are unable to evaluate it, then the result is considered *UNKNOWN*, *i.e*, the witness-validation procedure is unsuccessful in confirming the DepthK's verification result.

Validation of witness files became a rule in SV-COMP, as a way of deeply assessing verification results provided by a given verifier since it is possible to confirm, fully automatically, if values used in state-space exploration, which are available in a counterexample, lead to a correct result.

4.3.3 Invariant Generation using PIPS

PIPS aims to process large programs by performing a two-step analysis [113] automatically. Firstly, each program instruction is associated with an affine transformer, representing its underlying transfer function. Indeed, that is a bottom-up procedure, which starts from elementary instructions and then goes through compound statements, up to function definitions. Secondly, polyhedral invariants are propagated along with instructions, using previously computed transformers.

PIPS takes a C program as input, generates invariants, and prints a C program with those invariants as comments before each program statement. Then, we process those comments and instrument source code using assume statements[7].

Each comment must be preprocessed before being added to a C program, as those invariants generated by PIPS contain the suffix `#init` and includes mathematical expressions (*e.g.*, $2j < 5t$). For instance, Figure 4.3 shows transformers, preconditions, and generated syntax for the program described in [101], using PIPS. One may notice that those mathematical expressions do not contain a multiplication sign between constant and variable names, which does not consist in a valid C syntax.

```
//T() {0==-1}
void foo(float x){
// T(n){n==0}
 int n = 0;
// T(n){n#init==0}
    while(1)
  // T(n){n<=n#init+1}
        if(x)
  // T(n){n<=60, n<=n#init+1}
          if(n<60)
  // T(n){n==n#init+1, n<=60}
            n++;
          else
  // T(n){n==0, 60<=n#init}
            n = 0;
}
```

Figure 4.3 Sample with PIPS invariants using the structure #init.

In Algorithm 3, we describe the process of combining an original C program P with a set of invariants. As previously defined, P is the original program, $InvSet$ is the set of invariants, and P' is the combination of the original program P and invariants $InvSet$. The

[7]ESBMC requires assume statements to be written using `__ESBMC_assume(bool)`

complexity of that algorithm is $O(n^2)$, where n is the code size with invariants generated by PIPS. Algorithm 3 is split into three parts: (1) identification of structures `#init`, (2) generation of code to support translation of strutures `#init` into invariants, and (3) translation of the related mathematical expressions into ANSI-C code.

The first part of Algorithm 3 is performed in Line 35, which consists of reading each line of $InvSet$ and identifying whether a given comment is an invariant generated by PIPS (line 36) or not. If an invariant is identified and it contains a structure `#init`, then its location (*i.e.*, its line number) defined by *Inv.line* is stored, as well as type and name of the associated variable. After identifying structures `#init` in invariants, the second part of this algorithm analyzes each code line in $InvSet$, but now with the goal of identifying the beginning of each programming function (line 43). For each function that is identified, this algorithm checks whether it has structures `#init` (line 46) and, when that is true, a new code line is generated, for each related variable and at the beginning of the same function, with the declaration of an auxiliary variable, which contains a variable's old value, *i.e.*, its initial value. The newly created variable has the format `variable.type var_init = variable.name` , where `variable.type` is its identified type and `variable.name` is its identified (original) name.

Finally, each line containing a PIPS invariant is turned into expressions supported by the ANSI-C standard. Such a transformation consists of applying regular expressions (line 54) to multiplication operators (*e.g.*, from $2j$ to $2 * j$) and replacing structures `#init` by `_init`, which indicates that a new auxiliary variable must be generated and its content will be used as initial value for the original one. For each analyzed PIPS comment/invariant in $InvSet$, a new code line is generated in P'. The function `__ESBMC_assume`'s parameter is a conjunction of all invariants generated by PIPS.

4.3.4 Invariant Generation using PAGAI

PAGAI [76] is a static analyzer that uses structures of the low-level virtual machine (LLVM) compiler [97] and computes inductive invariants on numerical variables of an input program. Indeed, it uses a source-code analysis algorithm based on abstract interpretation to infer invariants for each control point in a C/C++ program. PAGAI performs a linear-relation analysis, which obtains invariants as convex polyhedra; however, it also supports other abstract domains, *e.g.*, octagons and products of intervals, which is not true for PIPS. Indeed, this last difference provides some variability for the adopted tools, which can be explored for tuned behaviour in specific scenarios.

In the experimental evaluation presented by Henry *et al.* [76], PAGAI was applied to real-world examples (industrial code and GNU programs). According to those authors, front-ends for many analysis tools place restrictions (*e.g.*, no backward goto instructions

Algorithm 3 The combination algorithm for PIPS.

Input: Program P, Set of invariants $InvSet$
Output: Program P'

```
if Inv is empty then
    return P
end
// dictionary to identify #init
dict_varinitloc[] ← { }
// copy comments of each invariant into array pips_comments
pips_comments[] ← extract_comments(Inv)
// list for the new code generated in the translation
P' ← { }
// Part 1 - identifying #init in the invariants
foreach Inv in InvSet do
    if pips_comments[Inv] is not empty then
        if pips_comments[Inv] has the pattern ([a-zA-Z0-9_]+)#init then
            dict_varinitloc[Inv.line] ← the variable suffixed #init
        end
    end
end
// list of translated invariants
listinvpips ← { }
// Part 2 - code generation to support #init structure and
   corrections regarding invariant format
foreach Inv in InvSet do
    P' ← Inv.line
    if P' at Inv.line is a function then
        if there exists some line in this function ∈ dict_varinitloc then
            foreach variable ∈ dict_varinitloc do
                P' ← Declare a variable as variable.type var_init=variable.name
            end
        end
    end
    // Part 3 - translation of the related mathematical expressions
       into ANSI-C code
    if pips_comments[Inv] is not empty then
        foreach expression ∈ pip_comments[Inv] do
            listinvpips ← Reformulate the expression according to the C programs syntax
              and replace #init by _init
        end
        P' ← __ESBMC_assume(conjunction of all invariants in listinvpips);
    end
end
return P'
```

and no pointer arithmetic), which may compromise safety-critical embedded programs. At the same time, PAGAI does not suffer from such issues. Nonetheless, it may apply coarse abstractions to some C/C++ programs, which can lead to weak invariants, whose conjunctions with safety properties are not inductive, as later confirmed in the experimental results of that work.

PAGAI takes a program as input, generates invariants, and outputs a new program, with invariants as comments before each statement. Similarly to our approach based on PIPS, those invariants are added to a program as assume statements. Let P denote the original program, $InvSet$ be a set that has each invariant extracted from the PAGAI annotations and its code location (*i.e.*, the line number of the code), in the analyzed program P, and P' be the combination between the original program P and invariants from $InvSet$. Aiming to include the program invariants inferred in $InvSet$, our PAGAI-based translation approach uses assume statements (in our case, `__ESBMC_assume`) to integrate them into P'. In contrast to PIPS, such invariants require minor changes, given that they are already ANSI-C compliant.

4.3.5 Illustrative Example

As an illustrative example, we describe a C program[8] extracted from SV-COMP 2019 [17] (shown in Figure 4.4). We chose this particular example because it demonstrates the importance of invariants, when verifying programs: the k-induction algorithm without invariants (using ESBMC) was unable to prove its correctness, during the same competition.

```
int __VERIFIER_nondet_int();
int main() {
  int offset, length, nlen;
  int i, j;
  for (i=0; i<nlen; i++) {
    for (j=0; j<8; j++) {
      assert(0 <= nlen-1-i);
      assert(nlen-1-i < nlen);
    }
  }
  return 0;
}
```

Figure 4.4 Verification example with two properties.

Two properties are being checked here (lines 7 and 8), eight times for each outer-loop iteration. One may notice that the nested loop does not change the properties being verified,

[8] *loop-invgen/id_build_true-unreach-call_true-termination.i*

but rather repeats checks eight times. Indeed, this is reduced by ESBMC, which checks a property only once per (outer) loop iteration, as follows:

1. `assert(nlen-1-i < nlen):`

   can be rewritten as `assert(i>=0);`

2. `assert(0 <= nlen-1-i):`

   can be rewritten as `assert(nlen >= i+1).`

This program is safe, so the traditional base case will never find a property violation. In order to prove correctness using the forward condition, a BMC tool will try to unwind the outer loop $2^{31} - 1$ times, while unwinding the inner one 8 times, on each iteration: only when it can unwind to that depth, it will reach all possible states, which is infeasibly expensive in both time and memory.

```
int __VERIFIER_nondet_int();
int main() {
  int offset, length, nlen;
  int i, j;
  i = 0;     // loop initial condition
  i = __VERIFIER_nondet_int();
// assume nondet
  __VERIFIER_assume(i<nlen);
// assume loop condition
  assert(0 <= nlen-1-i);
// loop body begin
  assert(nlen-1-i < nlen);  // ....
  i++;            // loop body end
  __VERIFIER_assume(!(i<nlen));
// assume negated loop cond
  return 0;
}
```

Figure 4.5 Verification example rewritten during the inductive step.

A plain inductive step is also unable to prove correctness, which is done by rewriting the mentioned program as illustrated in Figure 4.5. The transformations are: (1) all variables written inside a loop are treated as nondeterministic, as one can see in line 6, (2) it is assumed that the loop is indeed executed, as performed in line 7, and (3) after the loop's body, it terminates, which happens in line 11. One may notice that the inner loop is not present in this verification. Indeed, although `j` is written inside that loop body (when it is incremented), it is not part of the property's verification.

Indeed, the plain inductive step will easily find a counterexample for this program, as both `i` and `nlen` are unconstrained. ESBMC finds a property violation for `nlen = 4` and `i = -536870913`, and although `nlen = 4` is a value reachable by `nlen` during program execution, `i` will never reach `-536870913`. It is worth noticing that the related counterexample is spurious, due to the overapproximation previously mentioned.

```
int __VERIFIER_nondet_int();
int main()
{
  int offset, length, nlen = __VERIFIER_nondet_int();
  int i, j;
  for(i = 0; i <= nlen-1; i += 1) {
    __ESBMC_assume( 0<=i && i+1<=nlen );
    for(j = 0; j <= 7; j += 1) {
      __ESBMC_assume( 0<=i && i+1<=nlen && 0<=j && j<=7 );
      assert(0<=nlen-i-1);
      __ESBMC_assume( 0<=i && i+1<=nlen && 0<=j && j<=7 );
      assert(nlen-i-1<nlen);
    }
  }
  __ESBMC_assume( 0<=i && nlen<=i );
  return 0;
}
```

(a) Verification example with invariants generated by PIPS.

```
int __VERIFIER_nondet_int();
int main()
{
  int offset, length, nlen =__VERIFIER_nondet_int();
  int i, j;
  for (i=0; i<nlen; i++) {
    __ESBMC_assume( 2147483647-i >= 0 );
    __ESBMC_assume( i >= 0 );
    __ESBMC_assume( -1+nlen-i >= 0 );
    for (j=0; j<8; j++) {
      assert(0 <= nlen-1-i);
      assert(nlen-1-i < nlen);
    }
  }
  return 0;
}
```

(b) Verification example with invariants generated by PAGAI.

Figure 4.6 Verification example with invariants.

The programs in Figure 4.6 show invariants inserted by both PIPS and PAGAI, using the intrinsic function `__ESBMC_assume`. Both tools generate inductive invariants that are able to prove correctness of both properties:

1. PIPS generates `0<=i && i+1<=nlen`
2. PAGAI generates `i >= 0 && -1+nlen-i >= 0`,

Which are precisely the properties under verification. By assuming those two invariants, BMC tools will remove every state that violates those properties, leaving only reachable states. Our extended *k*-induction algorithm, combined with invariants, is a simple but substantial modification to the original *k*-induction and allows us to increase the number of programs that can be proved correct.

4.4 Experimental Evaluation

In this section, we present an experimental evaluation to establish a baseline for empirical comparisons involving DepthK, CPAchecker-*k*induction, and PDR-based tools, in the context of software verification, and to determine whether we can combine the strengths of *k*-induction with those of loop invariant generation. Our method was implemented in DepthK; we applied it to verify C benchmarks from embedded system applications and also the SV-COMP editions 2018[9] and 2019[10]. Additionally, we have also compared our approach against ESBMC [43], CBMC [95], and 2LS [32][11,12]. DepthK was also evaluated against the available PDR-based verifiers CPAchecker-CTIGAR [24], SeaHorn [73], and Vvt [72][13], which can be applied to actual C programs. SeaHorn is the most prominent software verifier that implements IC3; however, given its last participation in SV-COMP, in 2016, it did not perform well against other existing software verifiers (including DepthK), given a large number of incorrect results as reported by Beyer *et al.* [16]. Therefore, regarding IC3 tools comparison and given their current limited availability for C programs [19], we have considered only the verification tasks where the property to verify is the unreachability of a program location (*i.e.*, ReachSafety and SoftrwareSystems), as presented in Table 4.3.

[9] https://sv-comp.sosy-lab.org/2018/results/results-verified/
[10] https://sv-comp.sosy-lab.org/2019/results/results-verified/
[11] https://sv-comp.sosy-lab.org/2018/results/results-verified/META_ReachSafety_depthk.table.html
[12] https://sv-comp.sosy-lab.org/2018/results/results-verified/META_SoftwareSystems_depthk.table.html
[13] https://www.sosy-lab.org/research/pdr-compare/supplements/results/table.html

4.4.1 Experimental Setup

In order to evaluate the effectiveness of our proposed method, we use C programs with loops from public repositories, which constitute the main reference in the software verification area, such as: the set of C benchmarks from SV-COMP [17] and embedded systems applications [102, 131, 137]. For each benchmark, we check a single property encoded as an assertion or as an error location, i.e., we check whether an assertion is not violated or whether an error label is unreachable in any finite execution of the program.

The SV-COMP's benchmarks used in this experimental evaluation include:

- **ReachSafety**, which contains benchmarks for checking the reachability of an error location;
- **MemSafety**, which presents benchmarks for checking memory safety;
- **ConcurrencySafety**, which provides benchmarks for checking concurrency problems;
- **Overflows**, which is composed of benchmarks for checking if variables of signed-integers type overflow;
- **Termination**, which contains benchmarks for which termination should be decided;
- **SoftwareSystems**, which provides benchmarks from real software systems.

Regarding the PDR-based experimental results presented in Table 4.3, we consider only verification tasks that explore the strength of the PDR approach, where the property to verify is the unreachability of a program location. From the benchmarks above, we excluded properties for *overflows*, *memory safety*, and *termination*, which are not in the scope of this evaluation, and the categories *ReachSafety-Recursive* and *ConcurrencySafety*, each of which is not supported by at least one of the evaluated implementations. The remaining set of categories consists of 5591 verification tasks.

The embedded-system applications used in this experimental evaluation are classified in 3 categories: Powerstone [131], which is used for automotive-control and fax applications; Real-Time SNU [137], which contains a set of programs for matrix handling and signal processing, such as matrix multiplication and decomposition, second-degree equations, cyclic-redundancy check, Fourier transform, and JPEG encoding; and WCET [102], which is a set of programs for executing worst-case time analysis.

Here, we analyze the number of true and false positives and also the number of true and false negatives. Additionally, we generate a score for each analyzed tool based on the

total number of correct and incorrect results. In particular, this experimental evaluation is performed with the following tools:

- DepthK v3.1 with k-induction and invariants, using polyhedra (through PIPS and PAGAI), where ESBMC's parameters are defined in a wrapper script available in the DepthK's repository;
- ESBMC v6.0 along with plain k-induction, which is run with an interval-invariant generator that pre-processes input programs, infers invariants based on intervals, and includes them into programs.
- CBMC v5.11 with a bounded model checker, which, in the absence of additional loop transformations or k-induction, runs the script provided by Beyer, Dangl, and Wendler [20];
- CPAchecker[14] (revision 15596, acquired directly from its SVN repository) [20], which was executed with options k-induction together with invariants, and for k-induction without invariant;
- CPAchecker-CTIGAR (revision 27742) [24], which is an adaptation of PDR to software verification. Our evaluation compares CPAchecker with the implementations of CTIGAR;
- 2LS v0.7.2 along with k-induction and invariants, which is named kIkI and is executed with a wrapper script from SV-COMP 2019 [17];
- SeaHorn(F16-0.1.0-rc3) [73], which is a modular verification framework that uses constrained Horn clauses as the intermediate verification language. SeaHorn's verification condition generator is based on IC3;
- VVT [72], which is an implementation of the CTIGAR approach[24] that uses an SMT-based IC3 algorithm [28] incorporating Counterexample Guided Abstraction Refinement (CEGAR) [38]. Our evaluation compares the combination of Vvt-CTIGAR and bounded model checking, which is named as Vvt-Portfolio.

The present experimental evaluation was conducted on a computer with Intel Core $i7-4790$ CPU @ 3.60GHz and 32 GB of RAM, running Linux Ubuntu 18.04 LTS x64. Each verification task is limited to a CPU time of 15 minutes and a memory consumption of 15 GB.

[14]https://svn.sosy-lab.org/software/cpachecker/trunk

4.4.2 Experimental Results

After running all tools, we obtained the results shown in Tables 4.1 , 4.2 and 4.3.

Table 4.1 shows the results for the embedded system benchmarks, where **Tool** is the name of the Tool used in the experiments, **Correct Results** is the number of correctly proven programs, **Incorrect Results** is the number of programs where the respective Tool found at least one error, although being completely correct, or it does not identify an error, but the program contains a property violation, **Unknown and TO** is the number of programs that the Tool is unable to verify, due to lack of resources, tool failure (crash), or verification timeout (15 min), and **Time** is the runtime, in minutes, to verify the entire benchmark set.

Table 4.2 shows the results for all evaluated tools, regarding the SV-COMP 2019's benchmark suite, where **Category** is the SV-COMP's category, **Tool** is the name of the tool used in the experiments, **Correct True** is the number of programs where the respective tool did not find a bug, and that is correct, **Correct False** is the number of programs where the respective tool correctly found a bug, **True Incorrect** is the number of programs where the respective tool does not identify an error, which is correct, and **False Incorrect** is the number of programs where the respective tool found at least one error, although being completely correct.

Tool	DepthK (PIPS)	DepthK (PAGAI)	ESBMC (k-ind)	CPAchecker (k-ind)	CPAchecker (cont. ref. k-ind.)	CBMC (k-ind)	2LS
Correct Results	16	14	29	27	27	15	34
Incorrect Results	0	0	0	0	0	0	0
Unknown and TO	18	20	5	7	7	19	0
Time(min)	55.51	56.13	54.18	1.8	1.95	286.06	10.6

Table 4.1 Experimental results for the Powerstone, SNU, and WCET benchmarks.

Table 4.3 shows the results for all the 5591 verification tasks and has the same categorisation as Table 4.2. Nonetheless, it compares the effectiveness and efficiency of our implementation to the only available verifiers that implement a pure PDR approach for software-model checking.

Regarding Table 4.1, we have the following observations:

- The winning tool using k-induction is 2LS, which was able to give a correct result in all 34 verification tasks from the embedded-system benchmarks (Powerstone, SNU, and WCET): the difference with our approach is the way invariants are produced, i.e., we generate them before verification starts, while 2LS does that on each step, in order to continuously strengthen them.

- The results presented by DepthK using PIPS were lower than that of ESBMC and CPAchecker, but it outperformed CBMC with *k*-induction: although there exist no failures in our verification process, many inconclusive results were presented (i.e., unknown and TO), the reason being that both (PIPS and PAGAI) are not adequately prepared to handle some C features (e.g., bit-shift operations) used in embedded-system applications.

- The results presented by our approach are directly related to the maturity of each invariant generation tool: it is possible to improve the results presented by PIPS, since it has a wide variety of configurations that can be exploited by users, and PAGAI cannot be configured by users, which is the main difficulty for generating inductive invariants for embedded-system applications.

- Invariant generation configuration may be an improvement task: as already mentioned, PIPS presents a wide variety of configurations, which can be adaptively done for a given scenario or benchmark type.

- ESBMC v6.0 has received significant improvements in its *k*-induction algorithm, with bug and memory-leak fixes and guard simplification: using only its *k*-induction technique was enough to outperform our proposed method and its execution time was also shorter, due to implementation of those same improvements and because it does not require additional time for generating invariants;

- CPAchecker using only *k*-induction was slightly worse than ESBMC with *k*-induction. There exist two reasons to explain this result. The first one is that CPAchecker's invariant-generation algorithm works in the background, while it verifies a program. Consequently, the cumulative runtime of its two versions might exceed that of ESBMC, because the latter performs everything sequentially, in one single call, i.e., in order to generate a control flow graph, annotate a program with invariants, and finally verify it. The second one is that CPAchecker continuously refines invariants, which might run for longer times and then get strengthened, thus proving program correctness.

- CBMC with *k*-induction that verifies the absence of violated assertions under a given loop unwinding bound; however, it seems to be still experimental. In particular, CBMC relies on a limited loop unwinding technique and concurrent programs, which is unable to unroll nested loops [50].

In Tables 4.2, we have the following observations about those experimental results, per category.

Category	Tool	Correct True	Correct False	Total Correct	Incorrect True	Incorrect False	Total Incorrect
ReachSafety	DepthK (PAGAI)	62	700	762	0	4	4
	ESBMC (k-ind)	1332	893	**2225**	7	3	10
	2LS	1062	453	1515	1	2	3
	CBMC	641	669	1310	0	0	**0**
MemSafety	DepthK (PAGAI)	80	45	125	1	16	17
	ESBMC (k-ind)	130	64	**194**	8	2	10
	2LS	65	66	131	3	68	71
	CBMC	119	71	190	2	6	**8**
ConcurrencySafety	DepthK (PAGAI)	194	608	**802**	16	4	20
	ESBMC (k-ind)	182	600	782	14	7	21
	2LS	-	-	-	-	-	-
	CBMC	165	331	496	0	3	**3**
Overflows	DepthK (PAGAI)	0	167	167	0	0	**0**
	ESBMC (k-ind)	85	149	**234**	0	0	**0**
	2LS	87	140	227	0	0	**0**
	CBMC	31	169	200	0	0	**0**
Termination	DepthK (PAGAI)	266	0	266	14	0	14
	ESBMC (k-ind)	717	0	717	0	0	**0**
	2LS	676	306	**982**	0	3	3
	CBMC	718	0	718	0	0	**0**
SoftwareSystems	DepthK (PAGAI)	0	101	101	0	6	6
	ESBMC (k-ind)	1165	28	**1193**	12	2	14
	2LS	171	0	171	0	0	**0**
	CBMC	28	8	36	1	2	3
Total	DepthK (PAGAI)	602	1621	2223	31	30	61
	ESBMC (k-ind)	3611	1734	**5345**	41	14	55
	2LS	2061	965	3026	4	73	77
	CBMC	1702	1248	2950	3	11	**14**

Table 4.2 Experimental results for the SV-COMP'19.

Category	Tool	Correct True	Correct False	Total Correct	Incorrect True	Incorrect False	Total Incorrect
ReachSafety	DepthK (PAGAI)	395	644	1039	0	7	7
	CPAchecker-CTIGAR	397	214	611	0	1	**1**
	SeaHorn	1010	595	**1605**	6	105	111
	Vvt-Portfolio	528	311	839	9	22	31
SoftwareSystems	DepthK (PAGAI)	393	58	451	0	3	3
	CPAchecker-CTIGAR	435	41	477	0	0	**0**
	SeaHorn	1714	149	**1863**	40	12	52
	Vvt-Portfolio	0	0	0	0	0	0
Total	DepthK (PAGAI)	788	702	1490	0	10	10
	CPAchecker-CTIGAR	832	255	1087	0	1	**1**
	SeaHorn	2724	744	**3468**	46	117	163
	Vvt-Portfolio	528	311	839	9	22	31

Table 4.3 PDR-based experimental results SV-COMP'18.

- **ReachSafety**:
  - Our proposed method based on PAGAI presented the lowest number of correct answers, if compared with other existing approaches, because the generated invariants increased verification times and memory consumption rapidly, so DepthK did not reach a result promptly or even exceeded the amount of allowed memory;
  - ESBMC is the tool that correctly verified the most significant number of benchmarks, which demonstrated the effectiveness of the new interval-invariant used during verification processes. This improvement influenced results for some categories, such as *ReachSafety* and *SoftwareSystems*; however, ESBMC failed to verify other benchmarks, due to an internal bug in ESBMC, which made it unable to track variables going out of scope [43];
  - 2LS is the tool that correctly verified the second most significant number of benchmarks, which demonstrates its ability to analyze programs requiring combined reasoning about the shape and content of dynamic data structures and instrumentation for memory safety properties. Nonetheless, the reasoning about array contents is still missing, and the 2LS' algorithm *k*I*k*I does not support recursion yet.
- **MemSafety**:
  - ESBMC and CBMC were denoted as the first- and the second-best tools, respectively, due to recent improvements explicitly implemented for this category [95]. However, the number of incorrect results, although relatively low, is the main problem.
  - Our proposed method and 2LS were the tools that solved the lowest numbers of benchmarks. However, 2LS presented a large number of incorrect results, due to the lack of a bit-precise verification engine driven by weak invariants that did not take into account the nature of this category and its respective safety properties.
- **ConcurrencySafety**:
  - The proposed method using PAGAI is the tool that correctly verified the most significant number of benchmarks and was indeed able to increase that figure, due to its invariant inference. Also, our method was able to minimize ESBMC weakness by reducing the number of incorrect results. One of the main contributions of the present work is the generation of sufficiently inductive invariants

that can guide ESBMC to correct results, given that invariants inferred by PAGAI were essential to eliminate states that would typically induce ESBMC to fail.

- ESBMC has full concurrency support, and its standard context-BMC algorithm can solve a wide range of verification tasks, without the aid of *k*-induction and external invariants. Nonetheless, this tool produced the most significant number of incorrect results, due to the lack of support for some POSIX Pthreads functions, which are still un-modelled;
- CBMC was the tool that solved the lowest numbers of benchmarks, which was caused by current limitations in the treatment of pointers, and despite this tool's latest improvements [2];
- 2LS does not have native support for verifying concurrent programs based on POSIX/Pthreads [17].

- **Overflows**:
  - ESBMC and 2LS were denoted as the first- and the second-best tools, respectively, due to recent improvements regarding inductive invariants and considering programs that require joint reasoning about shape and content of dynamic data structures;
  - CBMC and our proposed method were the tools that solved the lowest numbers of benchmarks due to the lack of a bit-precise verification engine that efficiently handles arithmetic overflow checks.

- **Termination**:
  - 2LS and CBMC were the tools that successfully solved the highest number of benchmarks; however, if a larger number of unwindings is needed, the approach becomes quite inefficient. The strengths of BMC, on the other hand, are its predictable performance and amenability to the full spectrum of categories;
  - ESBMC using *k*-induction generated better results than our method that infers invariants, which happened because many inconclusive results were presented and that led to a meagre amount of verification tasks successfully analyzed;

- **SoftwareSystems**:
  - ESBMC using k-induction generated better results, when compared with other tools that infer invariants, which happened because of the relational analysis that can keep track of relations between variables;

- 2LS, which also uses inductive invariants, was slightly better than our method; however, it presented the same issues relating to limitation of states to be verified. That happened because some of the benchmarks, *e.g.* those requiring reasoning about arrays contents, demand invariants stronger than what is inferred by 2LS;
- DepthK overcame CBMC because it supports structures with pointers and considers variables of this type in the static analysis and also during invariant generation, as with 2LS.

Table 4.3 presents results for the chosen PDR-based tools, which were evaluated using the SV-COMP 2018's benchmark suite. There exist 5591 verification tasks, with 1457 of them containing bugs, while the remaining 4134 are considered to be safe, when checked with the best configurations of DepthK, CPAchecker, SeaHorn, and Vvt-portfolio. Indeed, such results give an overview of the best configurations used by the three different chosen software verifiers that use PDR. One may notice that SeaHorn achieved the highest numbers of total correct results, but it also presented a significant amount of total incorrect ones. Because SeaHorn is unsound for satisfiability, it can report that some expression-tree satisfies behavioural specification ψ, when in fact no such expression-tree exists. Also, SeaHorn verifier is unreliable for most bit-vector operations, *e.g.*, bit-shifting [81]. Despite considering only verification tasks that explore the strengths of the PDR approach, DepthK with PAGAI came second and overcame the other two PDR-based tools, as a result of our well-engineered implementation that provides invariants based on the idea of k-induction. Additionally and despite the PDR tools' compatibility restrictions, DepthK has widely used SV-benchmarks collection of verification tasks. In conclusion, our proposed method using PAGAI can be regarded as a competitive verification tool, when compared with the chosen PDR-based ones.

Figures 4.7 and 4.8 show the scoring system adopted in the SV-COMP's benchmarks, including comparative results for the SV-COMP's loops subcategory and embedded-system benchmarks, respectively. Here, we used only the SV-COMP's loops subcategory in this analysis, since loops are a widely recognized challenge regarding the inference of program invariants. One can notice that for embedded-system benchmarks, safe programs [43] were used, since the main goal here is to check whether strong (inductive) invariants are inferred, i.e., conditions that hold throughout an entire program, in order to prove correctness.

DepthK with PAGAI achieved a lower score than PIPS, in the embedded system benchmarks, due to the fact that PAGAI was unable to produce inductive invariants, with the potential to support ESBMC in reaching a verification result *true* or *false*. Indeed, PAGAI is a relatively new tool that is still under development, mainly regarding invariant prediction, and unlike PIPS that has many configuration options, PAGAI does not allow a combination of static analysis methods to infer invariants, which is the main reason for its low score.

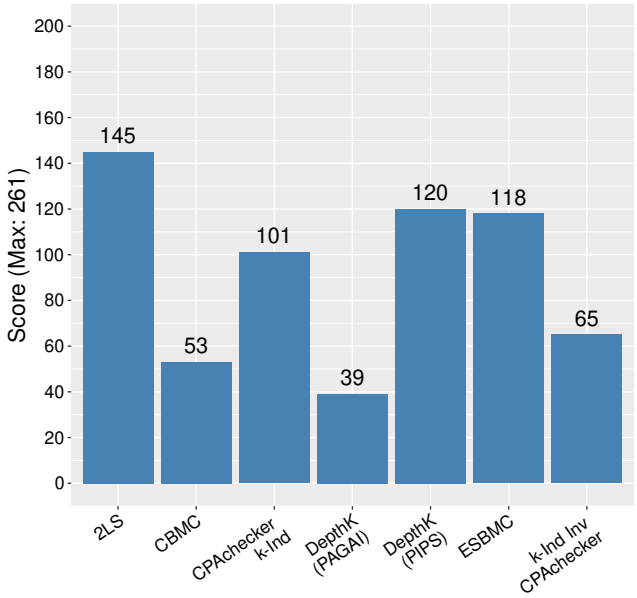

Figure 4.7 Score regarding loops subcategory.

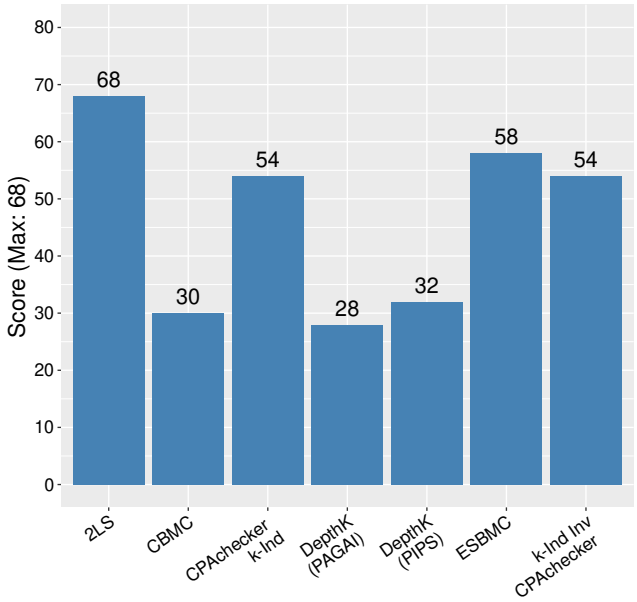

Figure 4.8 Score regarding embedded systems.

On the one hand, in the loops benchmarks, DepthK (PIPS) achieved the second-highest score, among all tools using invariants, being overcome only by 2LS. On the other hand, DepthK (PAGAI) presented the lowest score in the embedded-system benchmarks, mainly due to 58.82% of results identified as `Unknown`, i.e., when it is not possible to determine an outcome or due to tool failure. There exist also failures related to invariants and code

generation, which are given as input to the BMC procedure. As already mentioned, PAGAI is still under development (in a somewhat preliminary state), but one can argue that its results are still promising.

The CPAchecker *k*-induction is better than DepthK (PAGAI), since it is a more sophisticated tool while getting very close to ESBMC, in the embedded-system benchmarks. The main problem with both approaches is the number of errors that directly impacts their final scores. Nonetheless, CPAchecker *k*-induction was not better than 2LS, which can be explained by the fact that 2LS implements invariant generation techniques, incremental BMC, and *k*-induction. The 2LS's result was also the best regarding the embedded systems category, because it generated stronger invariants when compared with DepthK (PIPS)/PAGAI; however, the number of correct results was close to that obtained by ESBMC.

CBMC with *k*-induction was the tool that generated the highest number of inconclusive results, in the loops subcategory, and the verification time is noticeably superior to all tools used in the presented experimental evaluation; however, one can point out that the number of errors is not so different if compared with other approaches, and CBMC shows that it is as stable as the other tools.

In order to measure the impact of the invariants application to *k*-induction verification schemes, the distribution regarding DepthK + PIPS/PAGAI and ESBMC results were classified, per verification step: base case, forward condition, and inductive step. In this analysis, only the results related to DepthK and ESBMC were evaluated, given that they are part of the proposed approach and it is not possible to identify the steps of the *k*-induction algorithm (in standard logs), in other tools. Figure 4.9 shows the distribution of the results, for each verification step, regarding the SV-COMP's loops subcategory, while Figure 4.10 presents results for the embedded-system benchmarks.

The distribution of results in Figure 4.9, during the execution of the *k*-induction algorithm in the loops subcategory, shows that invariants generated by PIPS helped the *k*-induction algorithm in DepthK to increase the number of correct results. Here, the default ESBMC presents weaknesses for programs with loops, since it is unable to produce inductive loop invariants, in order to prove correctness.

By analyzing the presented results, we noticed that invariants allowed the *k*-induction algorithm to prove that loops were sufficiently unwound and whenever a property is valid for *k* unwindings, it is also valid after the next unwinding of a system. We also identified that the DepthK (PIPS) and DepthK (PAGAI) did not find a solution (leading to `Unknown` and `Timeout`) in 33.09% and 49.29% of the loops subcategory (see Figure 4.9), respectively. In the embedded system benchmarks, DepthK (PIPS) did not find a solution in 52.94% and DepthK (PAGAI) in 58.82% (see Figure 4.10). This is explained by the invariants generated

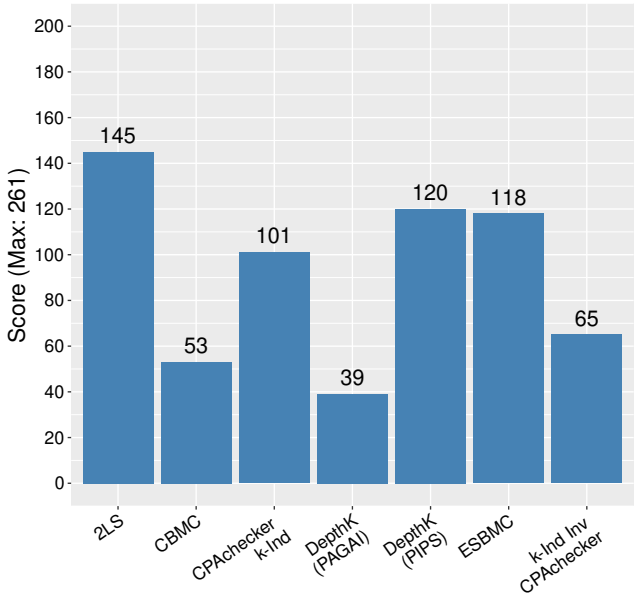

Figure 4.9 Results for the loops subcategory.

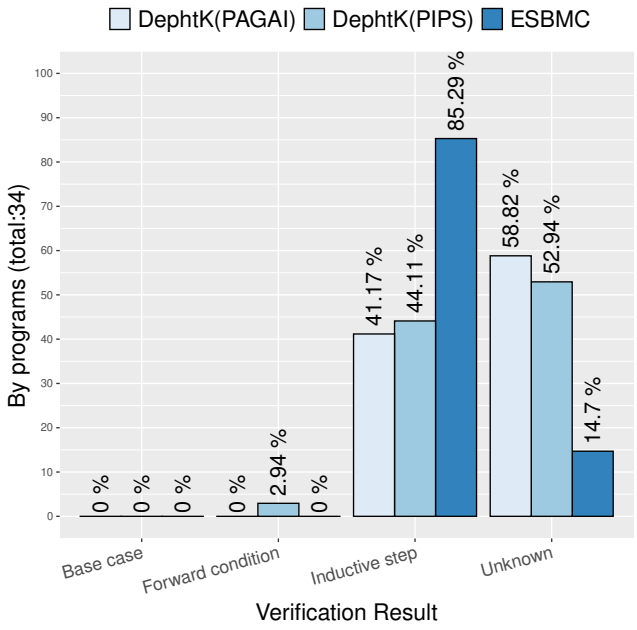

Figure 4.10 Results for the embedded programs.

from PIPS and PAGAI, which could not produce inductive invariants for the *k*-induction algorithm, either due to a transformer or invariants that are not convex.

We believe that PIPS' results can be significantly improved by fixing errors related to the tool's implementation since some results generated as `Unknown` are related to failures in our tool execution, which happened due to the algorithm that identifies functions and variables

in the analyzed source code. Additionally, we have identified that adjustments are needed in the PIPS's script parameters, in order to generate invariants, since PIPS has a broad set of commands for code transformation and that might lead to a positive impact, regarding invariant generation for specific classes of programs.

Due to the full range of benchmarks used in this experimental evaluation and the fact that PIPS has numerous configuration options, one possible improvement in our approach is to use PIPS, in order to allow us to discard unreachable states to reduce code, in addition to identifying loops to generate invariants to limit their unfolding. Thus, by adopting a new combination of PIPS's option set, results could be improved, and the *k*-induction algorithm would then speed up with stronger (inductive) invariants, for each benchmark.

4.5 Related Work

The *k*-induction algorithm has already been implemented and further extended by the software verification community, in many studies, which led to comparisons between our approach and other similar *k*-induction algorithms. Recently, Bradley et al. introduced the "property directed reachability" (or IC3) procedure for safety verification of systems [29, 74] and showed that IC3 could scale on certain benchmarks, where *k*-induction fails to succeed. We do not compare *k*-induction with IC3, as Bradley [29] already performed this, and focus on related *k*-induction procedures.

Donaldson et al. described a verification tool called Scratch [51], which can detect data races during direct memory access (DMA) in CELL BE processors from IBM [51], using *k*-induction. Properties (in the form of assertions) are automatically inserted to model behaviour of memory control-flow. The mentioned algorithm tries to find violations on those properties or prove that they hold indefinitely, by using a base case step and an inductive one, respectively, without checking the completeness threshold. That method also requires source code to be manually annotated with loop invariants, whereas our approach automatically generates and adds them to a given program. Finally, it can prove the absence of data races in several benchmarks, but it is restricted to verify a specific class of problems for a particular type of hardware. At the same time, our approach is evaluated over a more general group of programs, through (traditional) benchmarks from SV-COMP.

In another related work, Donaldson et al. described two tools for proving program correctness: K-Boogie and K-Inductor [50]. The former is an extension of the Boogie language, which aims to prove correctness (using *k*-induction) of programs written in some languages (Boogie, Spec, Dafny, Chalice, VCC, and Havoc), while the latter is a bounded model checker for C programs, which is built on top of CBMC [39]. Both use the *k*-induction

algorithm, which consists of a base case and an inductive step and, like previous work, the completeness threshold is not separately checked and relies only on the inductive step, in order to prove correctness. Their proposed *k*-induction has a preprocessing step; however, differently from ours, in which we introduce invariants, their preprocessing removes all nested loops and leaves only non-nested ones. Those authors compared the results of K-Inductor with Scratch. They showed that their new approach maintained the same coverage (in terms of correctly verified programs) while being faster. However, similarly to previous work, programs needed to be manually changed, in order to insert loop invariants, while our approach does it automatically. Madhukar et al. [100] described a method to accelerate the generation of program invariants, without *k*-induction, by adopting analysis of source code regarding loops. The basic idea was to identify invariants and their deviations in order to accelerate verification processes. Their article compared some model checkers (e.g., UFO [1], CPAchecker, CBMC, and IMPARA [25]), without using *k*-induction and regarding invariant generation, in order to speed up verification of programs with loops. In summary, the technique proposed by Madhukar et al. [100] is based on under-approximating loops for fast counterexample detection. It works as a pre-processor for replacing loops and improving verification processes, which reduces verification times and false results, in order to improve its confidence. In contrast to Madhukar et al., the proposed method does not modify existing loops, but instead includes assumptions based on invariants, in order to guide the *k*-induction algorithm.

Beyer et al. [20] introduced a different approach regarding invariant generation. In particular, they proposed a *k*-induction algorithm, which can generate invariants separately from the verification algorithm itself, named as Continuously Refined Invariants. The latter is an assistant for CPAchecker with *k*-induction and runs in parallel with verification tasks. It starts with weak invariants (i.e., without an invariant generator as PAGAI or PIPS), by using an abstract domain based on expressions over intervals. This method continuously adjusts and refines invariant precision, during verification processes, and creates inductive invariants [20]. This way, verification tasks can take advantage of previously generated values so that the *k*-induction verification algorithm can strengthen the induction hypotheses. In another work Beyer et al. [19] implemented a standalone PDR algorithm. In particular, they designed an invariant generator based on the ideas of PDR, and also evaluated the PDR invariant-generation module on an extensive set of C verification tasks. This method further extends the knowledge about PDR for software verification and outperforms an existing implementation of PDR. In summary, the PDR-based approach proposed here represents an effective and efficient technique for computing invariants, which are difficult to obtain with automated verification tools. However, the approach proposed by Beyer et al. solves

less verification tasks and takes longer than other verifiers, including DepthK. Additionally, our approach is evaluated over a more general group of programs, through the same set of benchmarks extracted from SV-COMP.

Brain et al. [32] proposed an incremental verification method called kIkI, which combines state-of-the-art verification approaches from literature (e.g., plain BMC, *k*-induction, and abstract interpretation), instead of using only *k*-induction algorithms. In particular, kIkI firstly applies the BMC technique to refute properties and then finds a counterexample: if it is not possible, a new verification procedure using *k*-induction (composed of the base case, forward condition, and inductive step) and invariants are applied, in order to prove that a program is safe. If the *k*-induction algorithm does not prove properties or does not generate a counterexample, an abstract interpretation technique, based on polyhedra, is applied, to generate invariants for the next state-space unrolling. In contrast to Brain et al. [32], the present work is based only on the application of *k*-induction, considering invariants in the polyhedral domain.

In another related work, Garg et al. [65] introduced the ICE-learning framework for synthesizing numerical invariants. According to Garg et al. [65], there exist many advantages in the (machine) learning approach. For instance, it typically concentrates on finding the simplest concept, which satisfies the constraints implicitly by providing a tactic for generalization. At the same time, white-box techniques (e.g., interpolation [96]) need to build in tactics to generalize. ICE-learning framework uses examples (test runs of the program on random inputs), counterexamples, and implications to refute the learner's conjectures. The ICE-algorithm iterates over all possible template formulas, thus growing in complexity, until it finds an appropriate formula, and adopts template-based synthesis techniques, which use constraint solvers. Garg et al. [65] use octagonal domain and present an empirical evaluation on benchmarks from SV-COMP loops category and programs from literature (e.g., [71]). In contrast to Garg et al. [65], we adopted the polyhedral domain and presented an extensive evaluation over different categories from SV-COMP benchmarks. For future work, we also plan to use a machine learning approach to infer invariants.

Ezudheen et al. [56] extended the ICE learning model for synthesizing invariants using Horn implication counterexamples [66]. According to Ezudheen et al. [56], their main contribution is to devise a decision tree-based Horn-ICE algorithm. The goal of the learning algorithm is to synthesize predicates, which are arbitrary Boolean combinations of the Boolean predicates and atomic predicates of the form $n \leq c$, where n denotes a numerical function, and where c is arbitrary. The implementation of the proposed method uses a predicate template of the form $x \pm y \leq c$, called octagonal constraints, where x, y are numeric program variables or non-linear expressions over numeric program variables and c is a

constant determined by the decision tree learner. Ezudheen et al. [56] present an evaluation using 109 programs, which 52 programs are from SV-COMP'18 recursive category. In comparison to Ezudheen et al. [56], we have used the polyhedral domain. However, our approach uses PAGAI to infer program invariants, where the abstract domains are provided by the APRON library [89], which include convex polyhedral, octagon, and products of intervals. We also, extend the experimental evaluation by adopting 6 categories from SV-COMP and embedded-system applications to validate the program invariant quality. Additionally, our approach does not need to produce samples or counterexamples to infer program invariants as Garg et al. [65] and Ezudheen et al. [56]. PIPS uses an interprocedural analysis, where each program instruction is associated with an affine transformer, representing its underlying transfer functions. For future work, we intend to investigate the strategy proposed in Ezudheen et al. [56], which chooses a template from a class of templates based on extracting features from a simple static analysis of the program and using priors gained from the experience of verifying similar programs in the past.

Champion et al. [36] proposed combining refinement types with the machine-learning-based for invariant discovery in ICE framework [65] suitable for higher-order program verification. Champion et al. [36] show the implementation of the proposed approach, which consists of two parts: (i) RType is a frontend (written in OCaml [142]) generating Horn clauses from programs written in a subset of OCaml; and (ii) HoIce, written in Rust [15], is one such Horn clause solver and implements the modified ICE framework. According to Champion et al. [36], RType supports a subset of OCaml including (mutually) recursive functions and integers, without algebraic data types. Champion et al. [36] argued that only considered programs that are safe since RType is not refutation-sound. Additionally, aiming to compare their Horn clause solver HoIce to other solvers, Champion et al. [36] show a comparison on the SV-COMP with Spacer (implemented in Z3). Where HoIce timeouts on a significant part of the benchmarks. Champion et al. [36] noted that are unsatisfiable; the ICE framework is not made to be efficient at proving unsatisfiability. In contrast to Champion et al. [36], our approach to infer invariants adopting PAGAI and PIPS that not apply machine-learning techniques. Related to the solver, the PAGAI uses Yices [54] or Z3 [47] through their C API, and PIPS is based on discrete differentiation and integration that is different from the usual abstract interpretation fixed-point computation based on widening. Champion et al. [36] show, in their experimental evaluation, that 11 programs fail because inherent limitations of the proposed approach, where two of them require an invariant of the form $x + y \geq z$. We argue that our proposed approach can handle with that form invariant since we adopt a polyhedral form such as $a.x + b.y \leq c$.

[15] https://www.rust-lang.org/

4.6 Conclusions

We described and evaluated a verification approach based on the *k*-induction proof rule, in which polyhedral abstraction of program behaviour is used to infer (inductive) invariants. The proposed method, which was implemented in a tool named as DepthK, was used to verify reachability properties using benchmarks from SV-COMP and embedded-systems automatically. In particular, 10522 verification tasks from SV-COMP 2019, 5591 verification tasks from SV-COMP 2018 and 34 ANSI-C programs from real-world embedded-system applications were evaluated. Also, a comparison among DepthK (using PIPS and PAGAI, as invariant generation tools), CPAchecker, ESBMC, CBMC, and 2LS, the latter with *k*-induction and invariants, was performed.

The DepthK's *k*-induction proof rule, together with invariants generated by PIPS, was able to provide verification results as accurate as those obtained with ESBMC *k*-induction, without invariant inference. We argue that the proposed method, in comparison to other existing software verifiers, shows promising results, which indicates that it can be sufficient to verify real programs. In particular, in benchmarks from SV-COMP 2019, DepthK was able to solve 2223 verification tasks and overcame other verifiers (*e.g.*, 2LS, CBMC and ESBMC) that use either BMC or *k*-induction proof rule in *ConcurrencySafety* category. Besides, DepthK was able to solve 1490 tasks and overcame other verifiers (*e.g.*, CPAchecker-CTIGAR and Vvt) that use PDR-based techniques.

Additionally, the combination of *k*-induction with invariants inferred by PAGAI led to fewer accurate results than those obtained with PIPS and also standard ESBMC *k*-induction. The configurations used in PIPS and PAGAI for benchmarks from embedded systems and SV-COMP indicate that our approach does not cover all possible categories of benchmarks. In particular, improvements in invariant generation must be made, depending on the program that is being verified. Those improvements range from refining PIPS and PAGAI configurations to deal with distinct verification conditions to the identification of the most suitable approach to be used for a given benchmark. As a result, DepthK would be able to deal with a series of verification tasks, through specific strategies both in the invariant generation and decision regarding the use of *k*-induction with invariant inference. Toward that, machine learning techniques could be used, which would be able to fine-tune configurations and guess the best approach to be employed [82].

For future work, we will investigate a hybrid approach to infer program invariants, which combines both PIPS and PAGAI, i.e., a strategy that merges invariants produced by both tools. We also aim to learn from counterexamples, in order to create stronger (inductive)

invariants, and as a consequence increase effectiveness from bug detection perspective using k-induction proof combined with invariants.

CHAPTER 5

Finding Security Vulnerabilities in Unmanned Aerial Vehicles Using Software Verification

Omar M. Alhawi, Mustafa A. Mustafa and Lucas C. Cordiro

Published: IEEE International Workshop on Secure Internet of Things - 2019

Statement of Contribution of joint Authorship[1]

Omar Alhawi (Principal Author)
Writing and compilation of manuscript, established methodology, experimental evaluation, preparation of tables and figures. Besides, presenting the paper in the workshop held in Luxembourg.

Mustafa Mustafa (Associate Supervisor)
Supervised and assisted with manuscript compilation, editing and co-author of manuscript.

Lucas Cordiro (Principal Supervisor)
Supervised and assisted with manuscript compilation, interpretation of results, editing and co-author of manuscript.

This Chapter is an exact copy of the workshop paper referred to above

[1] All authors read and approved the final manuscript.

Abstract

The proliferation of Unmanned Aerial Vehicles (UAVs) embedded with vulnerable monolithic software has recently raised serious concerns about their security due to concurrency aspects and fragile communication links. However, verifying security in UAV software based on traditional testing remains an open challenge mainly due to scalability and deployment issues. Here we investigate software verification techniques to detect security vulnerabilities in typical UAVs. In particular, we investigate existing software analyzers and verifiers, which implement fuzzing and bounded model checking (BMC) techniques, to detect memory safety and concurrency errors. We also investigate fragility aspects related to the UAV communication link. All UAV components (e.g., position, velocity, and attitude control) heavily depend on the communication link. Our preliminary results show that fuzzing and BMC techniques can detect various software vulnerabilities, which are of particular interest to ensure security in UAVs. We were able to perform successful cyber-attacks via penetration testing against the UAV both connection and software system. As a result, we demonstrate real cyber-threats with the possibility of exploiting further security vulnerabilities in real-world UAV software in the foreseeable future.

keywords UAV; Software Verification and Testing; Security;

5.1 Introduction

Unmanned Aerial Vehicles (UAVs), also sometimes referred to as *drones*, are aircrafts without human pilots on board; they are typically controlled remotely and autonomously and have been applied to different domains (e.g., industrial, military, and education). In 2018, PWC estimated the impact of UAVs on the UK economy, highlighting that they are becoming essential devices in various aspects of life and work in the UK. The application of UAVs to different domains are leading to GBP 42bn increase in the UK's gross domestic product and 628,000 jobs in its economy [120].

With this ever-growing interest also comes an increasing danger of cyber-attacks, which can pose high safety risks to large airplanes and ground installations, as witnessed at the Gatwick airport in the UK in late 2018, when unknown UAVs flying close to the runways caused disruption and cancellation of hundreds of flights due to safety concerns [121]. Recent studies conducted by the Civil Aviation Authority show that a 2kg UAV can cause a critical damage to a passenger jet windscreen [80]. Therefore, it remains an open question whether the *Confidentiality*, *Integrity*, and *Availability* (CIA) triad principles, which is a model designed to guide policies for information security [57], will be maintained during UAVs software development life-cycle. UAVs typically demand high-quality software to meet their target system's requirements. Any failures in embedded (critical) software, such as those embedded in avionics, might lead to catastrophic consequences in the real world. As a result, software testing and verification techniques are essential ingredients for developing systems with high *dependability* and *reliability* requirements, needed to guarantee both user requirements and system behavior.

Bounded Model Checking (BMC) was introduced nearly two decades ago as a verification technique to refute safety properties in hardware [22]. However, BMC has only relatively recently been made practical, as a result of significant advances in Boolean Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) [22]. Nonetheless, the impact of this technique is still limited in practice, due to the current *size* (e.g., number of lines of source code) and *complexity* (e.g., loops and recursions) of software systems. For instance, when a BMC-based verifier symbolically executes a program, it encodes all its possible execution paths into one single SMT formula, which results in a large number of constraints that need to be checked. Although BMC techniques are effective in refuting properties, they still suffer from the state-space explosion problem [62].

Fuzzing is a successful testing technique that can create a substantial amount of random data to discover security vulnerabilities in real-world software [104]. However, subtle bugs in UAVs might still go unnoticed due to the large state-space exploration, as recently reported

by Chaves et al. [37]. Additionally, according to Alhawi et al. [3], fuzzing could take a significant amount of time and effort to be completed during the testing phase of the software development life-cycle in addition to its code coverage issues. Apart from these limitations, fuzzing and BMC can enable a wide range of verification techniques. Some examples include automatic detection of bugs and security vulnerabilities, recovery of corrupt documents, patch generation, and automatic debugging. These techniques have been industrially adopted by large companies, including but not limited to Amazon Web Service (CBMC [42]), Microsoft (SAGE [68]), IBM (Apollo [8]), and NASA (Symbolic PathFinder [44]). For example, the SAGE fuzzer has already discovered more than 30 new bugs in large shipped Windows applications [68]. Nonetheless, an open research question consists of whether these techniques can be useful in terms of correctness and performance to verify UAV applications.

Our research investigates both fuzzing and BMC techniques to detect security vulnerabilities in real-world UAV software automatically. Thus, our main research goal is to allow the development of software systems, which are immune to cyber-attacks and thus ultimately improve software reliability. According to the current cyber-attacks profile concerning advanced UAVs, it becomes clear that the current civilian UAVs in the market are insecure even from simple cyber-attacks. To show this point of view, we highlight in our study real cyber-threats of UAVs by performing successful cyber-attacks against different UAV models. These cyber-attacks led to gain a full unauthorized control or cause the UAVs to crash. We show that pre-knowledge of the receptiveness of the UAV system components is all what attackers need to know during their reconnaissance phase before exploiting UAV weaknesses.

5.1.1 Contributions

Our main contribution is to propose a novel approach for detecting and exploiting security vulnerabilities in UAVs. We leverage the benefit of using both fuzzing and BMC techniques to detect security vulnerabilities hidden deep in the software state-space. In particular, we make three significant contributions:

- Provide a novel verification approach that combines fuzzing and BMC techniques to detect software vulnerabilities in UAV software.
- Identify different security vulnerabilities that UAVs can be susceptible to. We perform real cyber-attacks against different UAV models to highlight their cyber-threats.
- Evaluate a preliminary verification approach called “UAV fuzzer” to be compatible with the type of UAV software used in industry to exploit their vulnerabilities.

Although our current work represents an ongoing research, preliminary results show that fuzzing and BMC techniques can detect various software vulnerabilities, which are of particular interest to UAV security. We are also able to perform successful cyber-attacks via penetration testing against the UAV both connection and software system. As a result, we demonstrate real cyber-threats with the possibility of exploiting further security vulnerabilities in real-world UAV software in the foreseeable future.

5.1.2 Organisation

The rest of the paper is organised as follows. Section 5.2 describes the UAVs structure and the recent cyber attacks, in addition to other approaches used to verify security in UAVs. Section 5.3 introduces UAV software, UAV communication layer and the methodology to verify it using Fuzzing and Bounded Model Checking techniques. Section 5.4 then describes the benchmarks used and present the results to determine the effectiveness of our approach. Finally, Section 5.5 presents our conclusions.

5.2 Background

5.2.1 Generic Model of UAV Systems

Reg Austin [9] defines UAVs as a system comprising a number of subsystems, including the aircraft (often referred to as a UAV or unmanned air vehicle), its payloads, the Ground Control Station (GCS) (and, often, other remote stations), aircraft launch and recovery subsystems, where applicable, support subsystems, communication subsystems, and transport subsystems. UAVs have different shapes and models to meet the various tasks assigned to them, such as fixed-wing, single rotor, and multi-rotor, as illustrated in Fig. 5.1. However, their functional structure has a fixed standard, as shown in Fig. 5.2. Therefore, finding a security vulnerability in one model might lead to exploiting the same vulnerability in a wide range of different systems [48, 59].

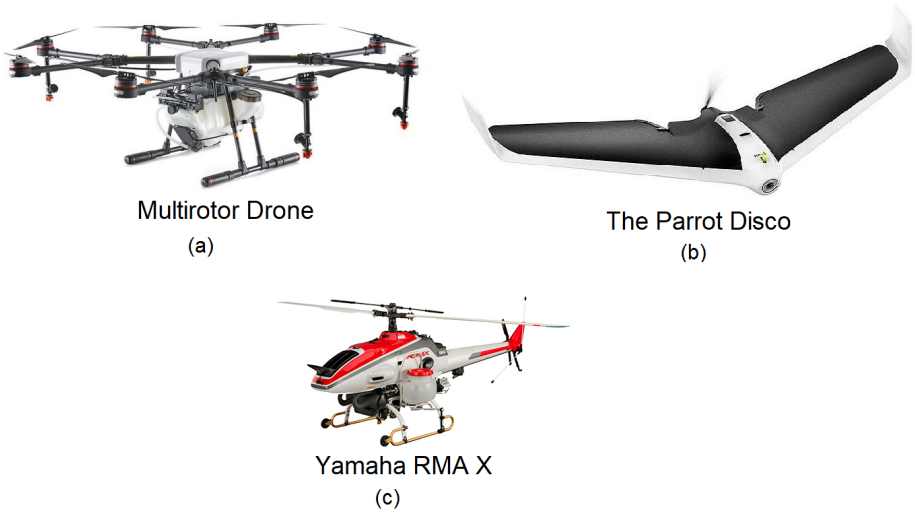

Figure 5.1 UAV types: multi-rotor (a), fixed wing (b), and single-rotor (c).

5.2.2 Cyber-Threats

A cyber-threat in UAVs represents a malicious action by an attacker with the goal of damaging the surrounding environment or causing financial losses, where the UAV is typically deployed [88]. In particular, with some of these UAVs available to the general public, ensuring their secure operation still remains an open research question, especially when considering the sensitivity of previous cyber-attacks described in the literature [90, 140].

One notable example is the control of deadly weapons as with the US military RQ-170 Sentinel stealth aircraft; it was intercepted and brought down by the Iranian forces in late

2011 during one of the US military operations over the Iranian territory [90]. In 2018, Israel released footage for one of its helicopters shooting down an Iranian replica model of the US hijacked drone [140].

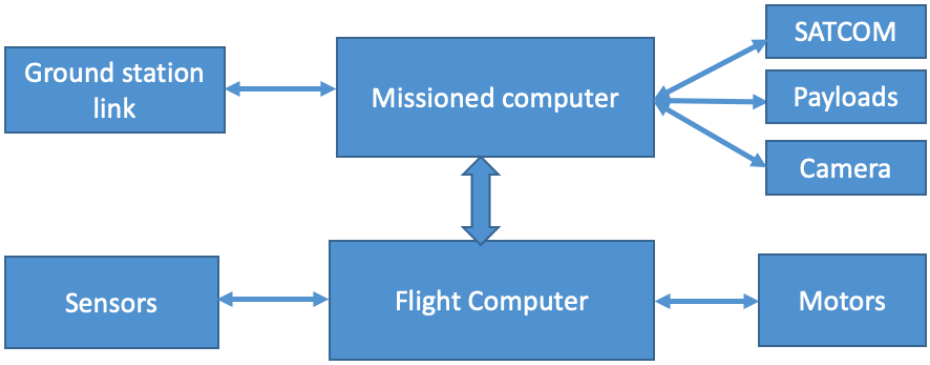

Figure 5.2 Functional structure of UAVs.

Further interest in UAV cyber-security has been raised following this attack. For example, Nils Rodday [127], a cyber-security analyst, was able to hack UAVs utilized by the police using a man-in-the-middle attack by injecting control commands to interact with the UAV. As a result of previous attacks, UAVs can be a dangerous weapon in the wrong hands. Obviously, cyber-attack threats exceeded the cyber-space barrier as observed by Tarabay, Lee and Frew [90, 140]. Therefore, enhancing the security and resilience of UAV software has become a vital homeland security mission, mainly due to the possible damage of cyber-attacks from the deployed UAVs.

5.2.3 Verification of Security in UAVs

The UAV components are typically connected together to enable secure and fast communication; if one component fails, the entire system can be susceptible to malicious attacks [9]. In this respect, various approaches have been taken to automatically verify the correctness of UAVs software. In particular, following the RQ-170 UAV accident in 2011, where Iran claimed hacking the sophisticated U.S. UAV [92], a group of researchers from the University of Texas proposed an usual exercise to the U.S. Department of Homeland Security; specifically simulated GPS signals were transmitted over the air from 620 m, where the spoofer induced the capture GPS receiver to produce position and velocity solutions, which falsely indicated the UAV position [119]. A similar study conducted in early 2018 performed a successful side-channel attack to leverage physical stimuli [110]. The authors were able to detect in real-time whether the UAV's camera is directed towards a target or not, by analyzing the encrypted communication channel, which was transmitted from a real UAV. These prior

studies were able to highlight GPS weaknesses, but they did not cover the UAV security issues w.r.t. all involved software elements, mainly when zero-day vulnerabilities are associated with the respective UAV outputs, i.e., a real UAV software bug that is unknown to the vendor responsible for patching or otherwise fixing the bug.

Other related studies focus on automated testing [45] and model-checking the behavior of UAV systems [37, 135]. For example, a verification tool named as Digital System Verifier (DSVerifier) [37] formally checks digital-system implementation issues, in order to investigate problems that emerge in digital control software designed for UAV attitude systems (i.e., software errors caused by finite word-length effects). Similar work also focuses on low-level implementation aspects, where Sirigineedi et al. [135] applied a formal modeling language called SMV to multiple-UAV missions by means of Kripke structures and formal verification of some of the mission properties typically expressed in Computational Tree Logic. In this particular study, a deadlock has been found and the trace generated by SMV has been successfully simulated. Note that these prior studies concentrate mainly on the low-level implementation aspects of how UAVs execute pilot commands. By contrast, we focus our approach on the high-level application of UAVs software, which is typically hosted by the firmware embedded in UAVs.

Despite the previously discussed limitations, BMC and Fuzzing techniques have been successfully used to verify the correctness of digital circuits, security, and communication protocols [135, 49]. However, given the current knowledge in ensuring security of UAVs, the combination of fuzzing and BMC techniques have not been used before for detecting security vulnerabilities in UAV software (e.g., buffer overflow, dereferencing of null pointers, and pointers pointing to unallocated memory regions). UAV software is used for mapping, aerial analysis and to get optimized images. In this study, we propose to use both techniques to detect security vulnerabilities in real-world UAV software.

5.3 Finding Software Vulnerabilities in UAVs Using Software Verification

5.3.1 Software In-The-Loop

UAV software has a crucial role to operate, manage, and provide a programmatic access to the connected UAV system. In particular, before a given UAV starts its mission, the missioned computer, as illustrated in Fig. 5.2, exports data required for this mission from a computer running the flight planning software. Then, the flight planning software allows the operator

to set the required flight zone (way-point mission engine), where the UAV will follow this route throughout its mission instead of using a traditional remote controller directly [45].

Dronekit[2] is an open-source software project, which allows one to command a UAV using Python programming language [141]; it enables the pilot to manage and direct control over the UAV movement and operation, as illustrated in Fig. 5.3, where one can connect to the UAV via a User Datagram Protocol (UDP) endpoint (line 3) with the goal of gaining control of the UAV by means of the "vehicle" object. In particular, UDP allows establishing a low-latency and loss-tolerating connection between the pilot and the UAV. This control process relies on the planning software inside the UAVs system, which in some cases the software might be permanently connected to the pilot controlling system (e.g., Remote Controller or GCS) due to live feedback or for real-time streaming.

```
1 from dronekit import connect
2 # Connect to UDP endpoint.
3 vehicle = connect('127.0.0.1:14550', wait_ready=True)
4 # Use returned Vehicle object to query device state:
5 print("Mode: %s" % vehicle.mode.name)
```

Figure 5.3 Python script to connect to a vehicle (real or simulated).

Our main research goal is to investigate in depth open-source UAVs code (e.g., DJI Tello[3] and Parrot Bebop[4]) to search for potential security vulnerabilities. For example, Fig. 5.4 shows a simple Python code to read and view various data status of Tello UAV. In particular, this Python code imports and defines the required libraries (lines from 1 to 3) and then connects the GCS to the UAV by using the predefined port and IP address in lines 11 and 12. As we can see from lines 15 to 25, the UAV will acknowledge the pilot commands and print the Tello current status. If an attacker is able to scan and locate the IP address that this particular UAV has used, then he/she would be able to easily intercept the data transmitted, inject a malicious code or take the drone out of service using a denial of service attack, which can lead the UAV to a crash, thus making it inaccessible. In order to detect potential security vulnerabilities in UAV software, we provide here an initial insight of how to combine BMC and fuzzing techniques with the goal of exploring the system state-space to ensure safe and secure operations of UAVs.

[2]https://github.com/dronekit/dronekit-python
[3]https://github.com/dji-sdk/Tello-Python
[4]https://github.com/amymcgovern/pyparrot

```
import socket
from time import sleep
import curses
INTERVAL = 0.2
...
local_ip = ''
local_port = 8890
socket=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
# socket for sending cmd
socket.bind((local_ip, local_port))
tello_ip = '192.168.10.1'
tello_port = 8889
tello_adderss = (tello_ip, tello_port)
socket.sendto('command'.encode('utf-8'), tello_adderss)
try:
  index = 0
  while True:
    index += 1
    response, ip = socket.recvfrom(1024)
    if response == 'ok':
      continue
    out = response.replace(';', ';\n')
    out = 'Tello State:\n' + out
    report(out)
    sleep(INTERVAL)
...
```

Figure 5.4 Python code fragment to read and view various data status of Tello UAV.

5.3.2 Illustrative Example Using UAV swarm

Throughout this paper, we use an illustrative example from UAV swarm, which consists of multiple UAVs to autonomously make decisions based on shared information; the safe and secure operation of multiple UAVs is particularly relevant since they have the potential to revolutionize the dynamics of conflict. In early 2019, we participated in a competitive-exercise[5], with five different UK universities; the main goal of this event consisted of teams from across the UK to compete against each other in a game of offense (red team) and defense (blue team) using swarms of UAVs, as illustrated in Fig. 5.5. As a result, this competition allowed us to highlight aspects of how to protect urban spaces from UAV swarms, which is a serious concern of modern society. This competition was sponsored by the British multinational defense, security, and aerospace company (BAE). Solutions developed by industry, such as the "jamming guns" and single "UAV catchers", fall short of what would be required to defend against a large automated UAV swarm attack.

For this particular illustrative example, using software verification and the UAV connection weakness, we were able to perform a successful cyber-attack against UAV models by scanning the radio frequencies and targeting the unwanted UAVs with just a raspberry-pi, a Linux OS installed on and 2.4 *GHz* antennas, as reported in our experimental evaluation.

[5]https://www.youtube.com/watch?v=dyyaY1VXqL4

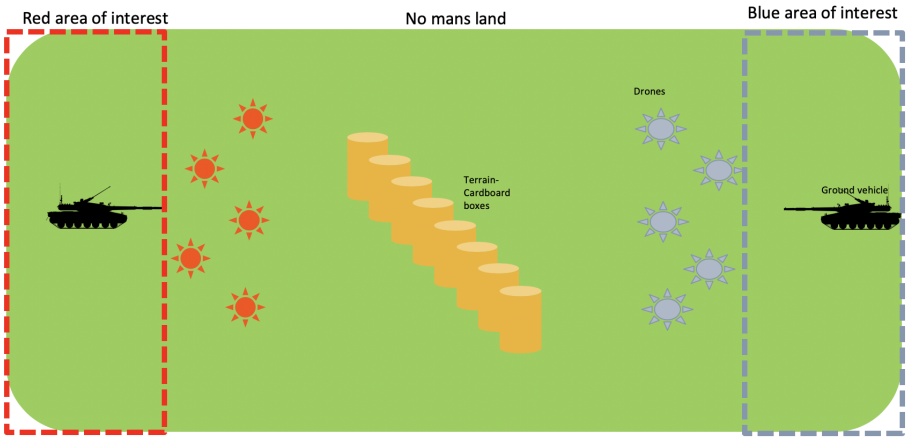

Figure 5.5 UAV Swarm Competition.

5.3.3 Verifying UAV Software using Fuzzing and BMC

We describe our novel verification approach approach called "UAV Fuzzer" to check for security vulnerabilities in UAVs. In particular, we check for user-specified assertions, buffer overflow, memory safety, division by zero, and arithmetic under- and overflow. Our verification approach consists of running a fuzzer engine using pre-collected test cases $TC = \{tc_1, tc_2, \ldots, tc_n\}$, where n represents the total number of pre-collected test cases, with the goal of initially exploring the state-space of the UAV software operation. Note that a test case tc_i used by our approach is similar to a real valid data, but it must contain a problem on it, or also called "anomalies". For example, to fuzz UAV software, a test case should be a connection between the UAV and GCS, so the mutated version generated of such a similar connection is called a test case tc.

In our "UAV Fuzzer", we keep track of each computation path of the program $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$, which represents the program unfolding for a bound k and a security property ϕ initially explored by our fuzzer. If our fuzzer engine gets stuck due to the mutations generated are not suited enough to the new state transition, our BMC tool runs against the target software to symbolically explore its uncovered state-space with the goal of checking the unexplored execution paths in Π of the UAV software. The idea behind BMC is to check the negation of a given property ϕ at a given depth k, i.e., given a transition system M, a property ϕ, and a limit of iterations k, BMC unfolds a given system k times and converts it into a Verification Condition (VC) ψ, such that ψ is *satisfiable* if and only if ϕ has a counterexample (cex) of depth less than or equal to k.

We formally describe our verification algorithms "UAV Fuzzer" by assuming that a given program P under verification is a state transition system M. In M, a state $s \in S$ consists of the

value of the program counter and the values of all program variables. A predicate $init_P(s)$ denotes that s is an initial state, $tr_p(s_i, s_j) \in T$ is a transition relation from s_i to s_j, $\phi(s)$ is the formula encoding for states satisfying a safety property, and $\psi(s)$ is the formula encoding for states satisfying a completeness threshold [94], which is equal to the maximum number of loop iterations occurring in P. For convenience, we define an error state ϵ, reachable if there exists a property violation in the program P. A counterexample cex^k is a sequence of states of length k from an initial state s_1 to ϵ. The main steps for our proposed verification algorithm are described in Algorithm 4.

Algorithm 4 UAV Fuzzer

1: Define pre-collected test cases $TC = \{tc_1, tc_2, \ldots, tc_n\}$ to be employed by the fuzzing engine.
2: Fuzzer engine begins to explore each execution path π_i starting from an initial state s and produces malformed inputs $I = \{\iota_1, \iota_2, \ldots, \iota_n\}$ to test for potential security vulnerabilities.
3: Store each π_i that has been verified and repeat step 2 until the fuzzer engine either reaches a crashing point or it cannot explore the next π_i in Π due to complex guard checks.
4: Run BMC to verify the remaining execution paths in Π that have not been previously explored by our fuzzer engine in steps 2 and 3.
5: Repeat step 4 until BMC falsifies or verifies ϕ or it exhausts time and memory limits.
6: Once BMC completely verifies the UAV code in step 5, it returns "false" if a property violation is found together with cex^k, "true" if it is able to prove correctness, or "unknown".

As an illustrative example, consider the code fragment shown in Fig. 5.4. First, our UAV fuzzer starts by defining new data based on the input expected by our targeted model (Tello UAV). As an example, Fig. 5.6 shows a valid test case generated by our UAV fuzzer, which is based on the module specification for the Tello UAV; this test case expects the IP address and specific port before launching the drone to start flying. Second, our UAV fuzzer engine starts exploring and running the UAV software (cf. Fig. 5.4) with the generated test-cases (cf. Fig. 5.6) and then it records each execution path π_i that has been explored. Third, when the UAV fuzzer engine reaches a complex condition and struggles to find its next path (line 5 of Fig. 5.6), UAV fuzzer will attempt to reconstruct the following path using BMC, which stores the current fuzzing transactions and restores the next path symbolically. Lastly, BMC will check for any further exception that occurs as a result of its execution. Additionally, UAV fuzzer can report the code coverage achieved during the testing process, and thus provide a better understanding of code coverage status.

```
1   while True:
2     index %= 1
3 # + replaced with %
4     response, ip = socket.recvfrom(1024)
5     if response == 'ok'
6       continue
```

Figure 5.6 Test case from the Tello UAV embedded software.

5.3.4 UAV Communication Channel

An UAV has a radio to enable and facilitate remote communication between the GCS and the UAV. In addition, it consists of different electronic components, which interact autonomously with a goldmine of data transmitted over the air during its flight's missions; this makes the communication channel in UAVs an ideal target for a remote cyber-threat. Therefore, ensuring secure (bug-free) software, together with a secure communication channel, emerges as a priority in successful deployment of any UAV system.

A successful false-data injection attack was demonstrated by Strohmeier et al. [139], which had devastating effects on the UAV system. The authors were able to successfully inject valid-looking messages, which are well-formed with reasonable data into the Automatic Dependent Surveillance-Broadcast (ADS-B) protocol. Note that this protocol is currently the only means of air traffic surveillance today, where Europe and US must comply with the mandatory use of this insecure protocol by the end of 2020 [139].

To investigate this layer further, we used Software-Defined Radio (SDR) system to receive, transmit, and analyze the UAV operational connection system (e.g., Ku-Band and WiFi). We have also investigated the information exchanged between UAV sensors and the surrounding environment for any potential security vulnerabilities (e.g., GPS Spoofing), as illustrated in Fig. 5.7. The signal that comes from the satellite is weak; hence, if an attacker uses a local transmitter under the same frequency, this signal would be stronger than the original satellite signal. As a result, the spoofed GPS-signal will override current satellite-signal, thereby leading to spoof a fake position for the UAV targeted. In this particular case, the UAV would then be hijacked and put in hold, waiting for the attacker's next command. Therefore, verifying the UAV software to build practical software systems with strong safety and security guarantees is highly needed in practice.

Our GPS spoofing attack is described in Algorithm 5. Here spoofer refers to a Full-Duplex device that is used to attack a particular UAV system to crash or take control of the UAVs. In our experimental evaluation with the UAV models DJI Tello and Parrot Bebop 2, we were able to perform a successful attack under 2.4 GHz to the target system. First, we were able to detect the drone frequency by one of the antennae, which configured to monitor the active 2.4

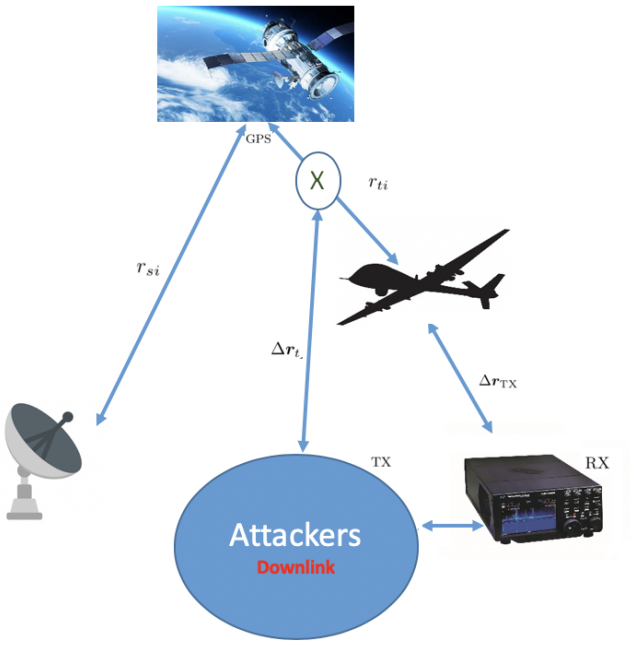

Figure 5.7 GPS-satellite-signal is overlaid by a spoofed GPS-signal.

GHz connection referred to as RX. The targeted drones are allocated based on identifying the drone default mac addresses owned by the Parrot company[6] and their unique SSID. Second, the other antenna was used to transmit the attacker new data referred to as TX. The distance between the attacker equipment and the target (r_{si} and r_{ti}) is vital for this attack, since the antenna/system strength and the delay caused are all taken into consideration during the attack.

Algorithm 5 GPS Spoofing Attack

1: The spoofer device should be located with the nominated antennas (i.e., 2.4 *GHz*).
2: The antenna configured on monitoring mode (*RX*) to detect the authentic signal from the available GPS satellites.
3: The vectors $\Delta\mathbf{r}_{TX}$ and $\Delta\mathbf{r}_t$ are for the antenna (*TX*) coordinates which distributed in a 3-dimensional array.
4: r_{si} and r_{ti} are the respective distances from the GPS satellite.

[6]http://standards-oui.ieee.org/oui/oui.txt

5.4 Preliminary Experimental Evaluation

We have performed a preliminary evaluation of our proposed verification approach to detect security vulnerabilities in UAVs. Our proposed method was implemented in a tool called DepthK[125], using BMC technique and invariant generators such as PIPS [114] and PAGAI [77]. PIPS is an inter-procedural source-to-source compiler framework for C and Fortran programs, PAGAI is a tool for automatic static analysis that is able to generate inductive invariants, both rely on a polyhedral abstraction of program behavior for infering invariants. We have evaluated the DepthK tool over a set of standard C benchmarks, which share common features of UAV code (e.g., concurrency and arithmetic operations). We have also evaluated our fuzzer engine to test a PDF software with the goal of checking its efficiency and efficacy to identify bugs. Lastly, we present our results in the swarm competition promoted by BAE systems.

5.4.1 Description of Benchmarks

The International Software Verification Competition (SV-COMP) [18], where DepthK participated, was run on a Linux Ubuntu 18.04 OS, 15 GB of RAM memory, a run-time limit of 15 minutes for each verification task and eight processing units of $i7-4790$ CPU. The SV-COMP's benchmarks used in this experimental evaluation include: *ReachSafety*, which contains benchmarks for checking reachability of an error location; *MemSafety*, which presents benchmarks for checking memory safety; *ConcurrencySafety*, which provides benchmarks for checking concurrency problems; *Overflows*, which is composed of benchmarks for checking whether variables of signed-integers type overflow; *Termination*, which contains benchmarks for which termination should be decided; *SoftwareSystems*, which provides benchmarks from real software systems. Our fuzzing experiments were ran on MacBook Pro laptop with 2.9 GHz Intel Core i7 processor and 16 GB of memory. We ran our fuzzing engine for at most of 12 hours for each single binary file. We analyzed and replayed the testing result after a crash was reported or after the fuzzer hit the time limit. To analyze the radio frequencies, we configured/compiled the required software for this purpose (e.g. bladerf, GQRX, OsmoSDR, and GNU Radio tool) using bladerf x40 device, ALFA high gain USB Wireless adapter and 2.4 *GHz* antennas. Additionally, we used the open-source UAVs code DJI Tello and Parrot Bebop.

5.4.2 Objectives

The impact of our study is a novel insight on the UAV security potential risks. In summary, our evaluation has the following three experimental questions to answer:

EQ1 **(Localization)** Can DepthK help us understand the security vulnerabilities that have been detected?

EQ2 **(Detection)** Can generational or mutational fuzzers be further developed to detect vulnerabilities in real-world software?

EQ3 **(Cyber-attacks)** Are we able to perform successful cyber-attacks in commercial UAVs?

5.4.3 Results

SV-COMP

Concurrency bugs in UAVs are one of the most difficult vulnerabilities to verify [135]. Our software verifier DepthK [125] has been used to verify and falsify safety properties in C programs, using BMC and k-induction proof rule techniques. In late 2018, we participated with the DepthK tool in SV-COMP 2019[7] against other software verifiers. Our verifier showed promising results over thousands of verification tasks, which are of particular interest to UAVs security (*e.g., Concurrency Safety* and *Overflows* categories), which answers **EQ1**.

Concurrency Safety category, which consists of 1082 benchmarks of concurrency problems, is one of the many categories verifiers run over; DepthK was able to accurately detect 966 problems from this category. For the *Overflows* category, which consists of 359 benchmarks for different signed-integers overflow bugs, DepthK was able to detect 167 problems. These results are summarized in Table 5.1. A task counts as *correct true* if it does not contain any reachable error location or assertion violation, and the tool reports "safe"; however, if the tool reports "unsafe", it counts as *incorrect true*. Similarly, a task counts as *correct false* if it does contain a reachable violation, and the tool reports "unsafe", together with a confirmed witness (path to failure); otherwise, it counts as *incorrect false* accordingly. Dirk Beyer [18] shows DepthK's results when compared with other verifiers in SV-COMP 2019.

Fuzzing Approach

According to a prior study [3], the generalizing fuzzing approach leads to a better result in discovering and recording software vulnerabilities compared with the mutational fuzzing

[7] https://sv-comp.sosy-lab.org/2019/

Table 5.1 DepthK Results in SV-COMP 2019.

Category list	Correct True	Correct False	Incorrect Results	Unknown
Concurrency Safety	194	772	20	96
Overflows	17	150	0	192

Table 5.2 Fuzzing Approaches Comparison.

Fuzzing Approaches	Target	Time	Faults
Generational Fuzzer	Sumatra PDF	45 hours	70
Mutational Fuzzer	Sumatra PDF	15 hours	23

approach if the test cases used in the fuzzing experiment are taken into account, which answers **EQ2**. Our experimental results applied to a PDF software called Sumatra PDF[8], which was chosen for evaluation purposes, are shown in Table 5.2. Here, the generational fuzzer was able to detect 70 faults in 45 hours in the Sumatra PDF, while the mutational fuzzer was able to detect 23 in 15 hours.

UAV Swarm Competition

As part of our participation at the UAV swarm competition sponsored by (BAE)[9], penetration testing was performed against the UAV both connection and software system, in which we were able to perform successful cyber-attacks, which answers **EQ3**. These attacks led to deliberately crash UAVs or to take control of different non-encrypted UAV systems (e.g., Tello and Parrot Bebop 2). This was achieved by sending connection requests to shut down a UAV CPU, thereby sending packets of data that exceed the capacity allocated by the buffer of the UAV's flight application and by sending a fake information packet to the device's controller. These results are summarized in Table 5.3, where we describe the employed UAV models and tools and whether we were able to obtain full control or crash. Note that due to the limitations of the competition, DepthK tool was not employed during the BAE competition; however, exploiting potential UAV software vulnerabilities is still a continuous research, where we intend to further exploit DepthK.

[8]https://www.sumatrapdfreader.org/free-pdf-reader.html

[9]https://www.cranfield.ac.uk/press/news-2019/bae-competition-challenges-students-to-counter-threat–from-uavs

Table 5.3 Results of the UAV Swarm Competition.

Vulnerability Type	UAV Model	Tool	Result
Spoofing	DJI Tello	Wi-Fi transmitter	Full Control
Denial of service			Full Control
Spoofing	Parrot bebop 2	Wi-Fi transmitter	Full Control
Denial of service			Crash

5.4.4 Threats to Validity

Benchmark selection: We report the evaluation of our approach over a set of real-world benchmarks, where the UAVs share the same component structure. Nevertheless, this set is limited within our research scope and the experiment results may not generalize to other models because other UAV models have a proprietary source-code. Additionally, we have not evaluated our verification approach using real UAV code written in Python, which is our main goal for future research.

Radio Spectrum: The frequencies we report on our evaluation were between 2.4 *GHz* and 5.8 *GHz*, as the two most common ranges for civilian UAVs; however, the radio regulations in the UK are complicated (e.g., we are required to be either licensed or exempted from licensing for any transmission over the air).

5.5 Conclusions and Future Work

Our ultimate goal is to develop a UAV fuzzer to be introduced as mainstream into the current UAV programming system, in order to build practical software systems robust to cyber-attacks. We have reported here an initial insight of our verification approach and some preliminary results using similar software typically used by UAVs. In order to achieve our ultimate goal, we have various tasks planned as follows:

- **Vulnerability Assessment**: Identify and implement simple cyber-attacks from a single point of attack against different UAV models. We will continue investigating Python vulnerabilities at the high-level system (e.g., UAV applications) and whether UAVs software is exploitable to those security vulnerabilities.

- **Python Fuzzer**: We will develop an automated python fuzzer by analyzing how to convert the UAV command packets into a fuzzing ones, in order to produce test cases, which are amenable to our proposed fuzzer.
- **GPS Analysis**: We identified based on numerical analysis on GPS, the cyber-attack UAVs might be vulnerable from. This investigation will continue to develop and simulate a GPS attack applied to a real UAV system.
- **Implementation**: Apply our proposed verification approach to test real-world software vulnerabilities, which can be implemented during the software development life-cycle to design a cyber-secure architecture.
- **Evaluation and Application**: Evaluate our proposed approach using real-world UAV implementation software. We will also compare our approach in different stages to check its effectiveness and efficiency.

Acknowledgment

Mustafa A. Mustafa is funded by the Dame Kathleen Ollerenshaw Fellowship awarded by The University of Manchester.

CHAPTER 6

Conclusion and Future Work

6.1 Conclusion

In essence, this study explored UAV's contemporary cyber challenge by asking whether software vulnerabilities are a credible threat to UAVs and the possibility to detect and verify. This work presents three contributions in the domain of UAV cyber-security research:

- Analysis of UAV software vulnerabilities
- Development of a new verification technique for vulnerability detection
- Demonstration of capability and usability of this method to detect various vulnerabilities

UAV software verification has already motivated other researchers to apply formal methods [129, 119]. However, verifying low-level software properties and UAV complex vulnerabilities such as concurrency was not accomplished until this work. Therefore, finding a practical approach to verify UAV software's correctness and detecting any potential vulnerabilities is essential to ensure UAV systems' security. As a result, fuzzing, BMC and *k*-induction proof rule techniques proposed to verify and detect software property violations. The investigation of UAV software vulnerabilities and the initial verification prototype considered this work's first achievement.

The proposed proof-by-induction algorithm combined *k*-induction with invariant inference and supported it with a context-bounded model checker engine to prove and refute software safety properties. The algorithm analysis demonstrated the proposed methodology's capability to verify UAV software from various possible vulnerabilities. The present experimental results show that $DepthK$ was able to verify various vulnerabilities (e.g, *overflows*, *memory safety*, *termination*, *ReachSafety* and *Concurrency*). Furthermore, the experimental outcomes show that UAV software might present failures after implementation. $DepthK$ employed

successfully to find property violations in real UAVs and verify the correctness of codes used in avionics embedded systems such as digital controller, signal processing functions, and JPEG encoding; results also showed that failures due to exploiting these violation caused notable changes in the UAV behaviour. Development of this method is the second achievement of this work, and its source code, as well as detailed documentation (including user and installation manual), are available online and this work was published as a peer-reviewed journal paper [5].

According to the provided verification results, $DepthK$ ranked first in identifying the most challenge UAV software vulnerabilities, such as concurrency bugs. It also outperforms other implementations that use k-induction with an interval-invariant generator and software verifiers that use plain BMC, and it is more precise than other PDR-based verifiers. Thus, our methodology demonstrated improvement over existing BMC and k-induction-based approaches and demonstrates its capability to use in a generic computing environment. Finally, $DepthK$ results are reproducible and reliable and can be used during the UAV software development life cycle to obtain more robust software implementations. Successfully employing the method and illustrating the tool capability was the third achievement of this work published as a peer-reviewed article [4].

6.2 Future Work

A various hybrid approach to infer program invariants can be implemented as future work. For example, a strategy that connected two distinct invariants to strengthen inductive invariants or learn from counterexamples to produce stronger invariants.

On the other hand, the design of new sophisticated UAV models and information related to its security measures is confidential; it is, therefore, difficult to determine which threats might affect the UAV system. What is certain is the presence of vulnerabilities in the UAV software and the fact that many 0-day vulnerabilities still to be found [144]. Further studies to investigate if an attacker can remotely exploit UAV software vulnerabilities through its communication system are potential future works.

Finally, $DepthK$ could use a machine learning approach to infer invariants, or by investigating the strategy Ezudheen *et al.* [56], proposed for choosing a template from a class of templates based on features extracted from a program's simple static analysis. In that sense, future invariant-generation tools will need new integration layers only when implemented, and as a result, it will improve the software testing performance in terms of both code coverage and bug finding ability.

Bibliography

[1] Albarghouthi, A., Gurfinkel, A., Li, Y., Chaki, S., and Chechik, M. (2013). UFO: Verification with interpolants and abstract interpretation. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *LNCS*, pages 637–640.

[2] Alglave, J., Kroening, D., and Tautschnig, M. (2013). Partial orders for efficient bounded model checking of concurrent software. In *Computer Aided Verification*, volume 8044 of *LNCS*, pages 141–157.

[3] Alhawi, O., Akinbi, A., and Dehghantanha, A. (2019a). Evaluation and Application of Two Fuzzing Approaches for Security Testing of IoT Applications. In *Handbook of Big Data and IoT Security*, pages 301–327. Springer, Cham.

[4] Alhawi, O., Mustafa, M., and Cordeiro, L. (2019b). Finding security vulnerabilities in unmanned aerial vehicles using software verification. In *IEEE International Workshop on Secure Internet of Things (SIoT)*. IEEE International Workshop on Secure Internet of Things (SIoT) , SIoT 2019 ; Conference date: 26-09-2019.

[5] Alhawi, O., Rocha, H., Gadelha, M. Y. R., Cordeiro, L., and Batista, E. (2020). Verification and refutation of c programs based on k-induction and invariant inference. *International Journal on Software Tools for Technology Transfer*, pages 1 – 21.

[6] Alur, R., Bodik, R., Juniwal, G., Martin, M. M., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. (2013). *Syntax-guided synthesis*. IEEE.

[7] Armando, A., Mantovani, J., and Platania, L. (2009). Bounded model checking of software using SMT solvers instead of SAT solvers. *Software Tools for Technology Transfer*, 11(1):69–83.

[8] Artzi, S., Kie zun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., and Ernst, M. D. (2008). Finding Bugs in Dynamic Web Applications. *ISSTA*, pages Pages 261–272.

[9] Austin, R. (2011). UNMANNED AIRCRAFT SYSTEMS UAVS DESIGN, DEVELOPMENT AND DEPLOYMENT. *A John Wiley and Sons*, 54.

[10] Authority, C. A. (2019). Drone and model aircraft registration and education service.

[11] Ball, T. and Rajamani, S. (2002). SLIC: A specification language for interface checking (of C). Technical report, Microsoft Research.

[12] Bansod, B., Singh, R., Thakur, R., and Singhal, G. (2017). A comparision between satellite based and drone based remote sensing technology to achieve sustainable development: A review. *Journal of Agriculture and Environment for International Development (JAEID)*, 111(2):383–407.

[13] Barrett, C., Sebastiani, R., Seshia, S., and Tinelli, C. (2009). *Handbook of Satisfiability*, chapter Satisfiability Modulo Theories, pages 825–885. IOS Press.

[14] Bessa, I. V., Ismail, H. I., Cordeiro, L. C., and Filho, J. a. E. (2016). Verification of fixed-point digital controllers using direct and delta forms realizations. *Des. Autom. Embedded Syst.*, 20(2):95–126.

[15] Beyer, D. (2015). Software verification and verifiable witnesses - (report on SV-COMP 2015). In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 9035 of *LNCS*, pages 401–416.

[16] Beyer, D. (2016). Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 9636 of *LNCS*, pages 887–904.

[17] Beyer, D. (2019a). Automatic verification of c and java programs: Sv-comp 2019. In Beyer, D., Huisman, M., Kordon, F., and Steffen, B., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 133–155, Cham. Springer International Publishing.

[18] Beyer, D. (2019b). Automatic verification of C and java programs: SV-COMP 2019. In *TACAS, LNCS 11429*, pages 133–155.

[19] Beyer, D. and Dangl, M. (2019). Software verification with PDR: implementation and empirical evaluation of the state of the art. *CoRR*, abs/1908.06271.

[20] Beyer, D., Dangl, M., and Wendler, P. (2015). Boosting k-induction with continuously-refined invariants. In *Computer-Aided Verification*, volume 9206 of *LNCS*, pages 622–640.

[21] Beyer, D. and Lemberger, T. (2017). Software verification: Testing vs. model checking. In *Haifa Verification Conference*, pages 99–114. Springer.

[22] Biere, A. (2009). *Handbook Of Satisfiability*, volume 185, chapter 26. IOS Press.

[23] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. (1999). Symbolic model checking without BDDs. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 1633 of *LNCS*, pages 193–207.

[24] Birgmeier, J., Bradley, A. R., and Weissenbacher, G. (2014). Counterexample to induction-guided abstraction-refinement (CTIGAR). In *Computer Aided Verification*, volume 8559 of *LNCS*, pages 831–848.

[25] Björn Wachter, D. K. and Ouaknine, J. (2013). Verifying multithreaded software with impact. In *Formal Methods in Computer Aided Design*, pages 210–217.

[26] Bjørner, N. and Gurfinkel, A. (2015). Property directed polyhedral abstraction. In D'Souza, D., Lal, A., and Larsen, K. G., editors, *Verification, Model Checking, and Abstract Interpretation*, pages 263–281, Berlin, Heidelberg. Springer Berlin Heidelberg.

[27] Bradley, A. R. (2011a). SAT-based model checking without unrolling. In *Verification, Model Checking, And Abstract Interpretation*, volume 6538 of *LNCS*, pages 70–87.

[28] Bradley, A. R. (2011b). Sat-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer.

[29] Bradley, A. R. (2012a). IC3 and beyond: Incremental, inductive verification. In *Computer Aided Verification*, volume 7358 of *LNCS*, page 4.

[30] Bradley, A. R. (2012b). Understanding IC3. In *Theory and Applications of Satisfiability Testing*, volume 7317 of *LNCS*, pages 1–14.

[31] Bradley, A. R. and Manna, Z. (2007). *The Calculus Of Computation: Decision Procedures With Applications To Verification*. Springer-Verlag New York, Inc., 1st edition.

[32] Brain, M., Joshi, S., Kroening, D., and Schrammel, P. (2015). Safety verification and refutation by k-invariants and k-induction. In *Static Analysis Symposium*, volume 9291 of *LNCS*, pages 145–161.

[33] Brat, G. and Jonsson, A. (2005). Challenges in verification and validation of autonomous systems for space exploration. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 5, pages 2909–2914. IEEE.

[34] Carlson, B. J. (2001). *PAST UAV PROGRAM FAILURESAND IMPLICATIONS FOR CURRENT UAV PROGRAMS*. Air Command and Staff College.

[35] Carter, M., He, S., Whitaker, J., Rakamarić, Z., and Emmi, M. (2016). SMACK software verification toolchain. In *International Conference on Software Engineering*, pages 589–592.

[36] Champion, A., Chiba, T., Kobayashi, N., and Sato, R. (2018). Ice-based refinement type discovery for higher-order functional programs. In *24th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS)*, pages 365–384.

[37] Chaves, L., Bessa, I., Ismail, H., Frutuoso, A., Cordeiro, L., and de Lima Filho, E. (2018). DSVerifier-Aided Verification Applied to Attitude Control Software in Unmanned Aerial Vehicles. *IEEE Transactions on Reliability*, 67.

[38] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2000a). Counterexample-guided abstraction refinement. In *Computer-Aided Verification*, volume 1855 of *LNCS*, pages 154–169.

[39] Clarke, E., Kroening, D., and Lerda, F. (2004). A Tool For Checking ANSI-C Programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176.

[40] Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263.

[41] Clarke, E. M., Grumberg, O., and Peled, D. A. (2000b). *Model Checking*. MIT Press, Cambridge, MA, USA.

[42] Cook, B., Khazem, K., Kroening, D., Tasiran, S., Tautschnig, M., and Tuttle, M. R. (2018). Model checking boot code from AWS data centers. In *Computer Aided Verification*, volume 10982, pages 467–486. Springer.

[43] Cordeiro, L. C., Fischer, B., and Marques-Silva, J. (2012). SMT-Based Bounded Model Checking For Embedded ANSI-C Software. *IEEE Transactions on Software Engineering*, 38(4):957–974.

[44] Corina S. Pasareanu (2010). Using Symbolic (Java) PathFinder at NASA. *Nasa*.

[45] Day, M. A., Clement, M. R., Russo, J. D., Davis, D., and Chung, T. H. (2015). Multi-uav software systems and simulation architecture. In *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 426–435.

[46] de la Cámara, P., Castro, J. R., Gallardo, M. d. M., and Merino, P. (2011). Verification support for arinc-653-based avionics software. *Software Testing, Verification and Reliability*, 21(4):267–298.

[47] De Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT solver. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 4963 of *LNCS*, pages 337–340.

[48] Dey, V., Pudi, V., Chattopadhyay, A., and Elovici, Y. (2018). Security vulnerabilities of unmanned aerial vehicles and countermeasures: An experimental study. In *VLSID*, pages 398–403. IEEE Computer Society.

[49] Domin, K., Symeonidis, I., and Marin, E. (2016). Security analysis of the drone communication protocol: Fuzzing the mavlink protocol.

[50] Donaldson, A., Haller, L., Kroening, D., and Rümmer, P. (2011a). Software verification using k-induction. In *Static Analysis Symposium*, volume 6887 of *LNCS*, pages 351–368.

[51] Donaldson, A., Kroening, D., and Rümmer, P. (2011b). SCRATCH: A tool for automatic analysis of DMA races. In *Symposium On Principles And Practice Of Parallel Programming*, pages 311–312.

[52] Donaldson, A. F., Haller, L., and Kroening, D. (2011c). Strengthening induction-based race checking with lightweight static analysis. In *Verification, model checking, and abstract interpretation*, volume 6538 of *LNCS*, pages 169–183.

[53] Donaldson, A. F., Kroening, D., and Ruemmer, P. (2011d). Automatic analysis of DMA races using model checking and k-induction. *Formal Methods in System Design*, 39(1):83–113.

[54] Dutertre, B. (2014). Yices 2.2. In *Computer-Aided Verification*, volume 8559 of *LNCS*, pages 737–744.

[55] Eén, N. and Sörensson, N. (2003). Temporal Induction By Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560.

[56] Ezudheen, P., Neider, D., D'Souza, D., Garg, P., and Madhusudan, P. (2018). Horn-ice learning for synthesizing invariants and contracts. *Proc. ACM Program. Lang.*, 2(OOPSLA):131:1–131:25.

[57] Farooq, M., Waseem, M., Khairi, A., and Mazhar, S. (2015). Article: A critical analysis on the security concerns of internet of things (iot). *International Journal of Computer Applications*, 111(7):1–6. Full text available.

[58] Ferrando, A., Dennis, L. A., Ancona, D., Fisher, M., and Mascardi, V. (2018). Verifying and validating autonomous systems: Towards an integrated approach. In Colombo, C. and Leucker, M., editors, *Runtime Verification*, pages 263–281, Cham. Springer International Publishing.

[59] Frei, S., May, M., Fiedler, U., and Plattner, B. (2006). Large-scale vulnerability analysis. In *LSAD '06*, pages 131–138.

[60] Fu, H., Wang, Z., Chen, X., and Fan, X. (2018). A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques. *Software Quality Journal*, 26(3):855–889.

[61] Furia, C. A., Meyer, B., and Velder, S. (2014). Loop invariants: Analysis, classification, and examples. *ACM Comput. Surv.*, 46(3).

[62] Gadelha, M. R., Monteiro, F. R., Morse, J., Cordeiro, L. C., Fischer, B., and Nicole, D. A. (2018a). ESBMC 5.0: an industrial-strength C model checker. In *ASE*, pages 888–891. ACM Press.

[63] Gadelha, M. Y. R., Ismail, H. I., and Cordeiro, L. C. (2017). Handling loops in bounded model checking of C programs via k-induction. *Software Tools for Technology Transfer*, 19(1):97–114.

[64] Gadelha, M. Y. R., Monteiro, F. R., Cordeiro, L. C., and Nicole, D. A. (2018b). Towards counterexample-guided k-induction for fast bug detection. In *ACM Joint European Software Engineering Conference And The Foundations Of Software Engineering*, pages 765–769.

[65] Garg, P., Löding, C., Madhusudan, P., and Neider, D. (2014). ICE: A robust framework for learning invariants. In *26th International Conference Computer Aided Verification (CAV)*, pages 69–87.

[66] Garg, P., Neider, D., Madhusudan, P., and Roth, D. (2016). Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 499–512.

[67] Gat, E. (2004). Autonomy software verification and validation might not be as hard as it seems. In *2004 IEEE Aerospace Conference Proceedings (IEEE Cat. No.04TH8720)*, volume 5, pages 3123–3128 Vol.5.

[68] Godefroid, P., Levin, M. Y., and Molnar, D. (2012). Automated Whitebox Fuzz Testing. *Queue - Networks*.

[69] Goldberg, D. (1991). What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48.

[70] Große, D., Le, H., and Drechsler, R. (2009). Induction-based formal verification of systemC TLM designs. In *Workshop On Microprocessor Test And Verification*, pages 101–106.

[71] Gulavani, B. S., Henzinger, T. A., Kannan, Y., Nori, A. V., and Rajamani, S. K. (2006). Synergy: A new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 117–127. ACM.

[72] Günther, H., Laarman, A., and Weissenbacher, G. (2016). Vienna verification tool: IC3 for parallel software. In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 9636 of *LNCS*, pages 954–957.

[73] Gurfinkel, A., Kahsai, T., Komuravelli, A., and Navas, J. A. (2015). The seahorn verification framework. In *Computer-Aided Verification*, volume 9206 of *LNCS*, pages 343–361.

[74] Hassan, Z., Bradley, A. R., and Somenzi, F. (2013). Better generalization in IC3. In *Formal Methods In Computer-Aided Design*, pages 157–164.

[75] Heizmann, M., Christ, J., Dietsch, D., Ermis, E., Hoenicke, J., Lindenmann, M., Nutz, A., Schilling, C., and Podelski, A. (2013). Ultimate automizer with smtinterpol. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *LNCS*, pages 641–643.

[76] Henry, J., Monniaux, D., and Moy, M. (2012a). PAGAI: A path sensitive static analyser. *Electronic Notes in Theoretical Computer Science*, 289:15–25.

[77] Henry, J., Monniaux, D., and Moy, M. (2012b). PAGAI: A Path Sensitive Static Analyser. In *Electron. Notes Theor. Comput. Sci.*, pages 15–25. Elsevier Science Publishers B. V.

[78] Henzinger, T. A., Pei-Hsin Ho, and Wong-Toi, H. (1995). Hytech: the next generation. In *Proceedings 16th IEEE Real-Time Systems Symposium*, pages 56–65.

[79] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

[80] House, A. (2018). *Drone Safety Risk: An assessment*. Civil Aviation Authority.

[81] Hu, Q., Breck, J., Cyphert, J., D'Antoni, L., and Reps, T. (2019). Proving unrealizability for syntax-guided synthesis. In Dillig, I. and Tasiran, S., editors, *Computer Aided Verification*, pages 335–352, Cham. Springer International Publishing.

[82] Hutter, F., Babic, D., Hoos, H. H., and Hu, A. J. (2007). Boosting verification by automatic tuning of decision procedures. In *Formal Methods in Computer-Aided Design*, pages 27–34.

[83] Hwang, I., Kim, S., Kim, Y., and Seah, C. E. (2010). A survey of fault detection, isolation, and reconfiguration methods. *IEEE Transactions on Control Systems Technology*, 18(3):636–653.

[84] IEEE (2008). *IEEE Standard For Floating-Point Arithmetic*. IEEE 754-2008.

[85] Ismail, H. I., Bessa, I. V., Cordeiro, L. C., Lima Filho, E. B., and Chaves Filho, J. a. E. (2015). Dsverifier: A bounded model checking tool for digital systems. In *Proceedings of the 22nd International Symposium on Model Checking Software - Volume 9232*, SPIN 2015, page 126–131, Berlin, Heidelberg. Springer-Verlag.

[86] Ivančić, F., Shlyakhter, I., Gupta, A., and Ganai, M. K. (2005). Model Checking C Programs Using F-SOFT. *ICCD*, pages 297–308.

[87] Jacklin, S., Schumann, J., Gupta, P., Lowry, M., Bosworth, J., Zavala, E., Hayhurst, K., Belcastro, C., and Belcastro, C. (2004). Verification, validation, and certification challenges for adaptive flight-critical control system software. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, page 5258.

[88] Javaid, A. Y., Sun, W., Devabhaktuni, V. K., and Alam, M. (2012). Cyber security threat analysis and modeling of an unmanned aerial vehicle system. In *HST*, pages 585–590.

[89] Jeannet, B. and Miné, A. (2009). Apron: A library of numerical abstract domains for static analysis. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV'09, pages 661–667, Berlin, Heidelberg. Springer-Verlag.

[90] Joanna Frew (2018). An overview of new armed drone operators The Next Generation. Technical report, Drone Wars UK, Oxford.

[91] Jovanović, D. and Dutertre, B. (2016). Property-directed k-induction. In *Formal Methods In Computer-Aided Design*, pages 85–92.

[92] Kerns, A. J., Shepard, D. P., Bhatti, J. A., and Humphreys, T. E. (2014). Unmanned aircraft capture and control via gps spoofing. *Journal of Field Robotics*, 31(4):617–636.

[93] Kim, B. U. and Humphrey, L. R. (2014). Satisfiability checking of ltl specifications for verifiable uav mission planning. In *52nd Aerospace Sciences Meeting*, page 0793.

[94] Kroening, D., Ouaknine, J., Strichman, O., Wahl, T., and Worrell, J. (2011). Linear Completeness Thresholds For Bounded Model Checking. In *Computer-Aided Verification*, volume 6806 of *LNCS*, pages 557–572.

[95] Kroening, D. and Tautschnig, M. (2014). CBMC - C bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *LNCS*, pages 389–391.

[96] Kroening, D. and Weissenbacher, G. (2011). Interpolation-based software verification with wolverine. In *23rd International Conference Computer Aided Verification (CAV)*, pages 573–578.

[97] Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *Symposium On Code Generation And Optimization*, pages 75–96.

[98] Le Lann, G. (1996). The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems. Research Report RR-3079, INRIA. Projet REFLECS.

[99] Lipka, R., Paška, M., and Potužák, T. (2014). Simulation testing and model checking: A case study comparing these approaches. In *International Workshop on Software Engineering for Resilient Systems*, pages 116–130. Springer.

[100] Madhukar, K., Wachter, B., Kroening, D., Lewis, M., and Srivas, M. K. (2015). Accelerating invariant generation. In *Formal Methods in Computer-Aided Design*, pages 105–111.

[101] Maisonneuve, V., Hermant, O., and Irigoin, F. (2014). Computing invariants with transformers: Experimental scalability and accuracy. In *Numerical and Symbolic Abstract Domains*, pages 17–31.

[102] Mälardalen WCET Research Group (2012). WCET benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html. [Online; accessed August-2019].

[103] Merz, F., Falke, S., and Sinz, C. (2012). LLBMC: Bounded Model Checking Of C And C++ Programs Using A Compiler IR. In *VSTTE*, volume 7152 of *LNCS*, pages 146–161.

[104] Miller, B. P., Cooksey, G., and Moore, F. (2006). An empirical study of the robustness of macos applications using random testing. In *Proceedings of the 1st International Workshop on Random Testing, RT*, pages 46–54. ACM.

[105] Ministry of Transport, Ministry of Business, I., and Employment (2019). *Drones: Benefits study*. Ministry of Transport.

[106] Mohamed, N., Al-Jaroodi, J., Jawhar, I., and Lazarova-Molnar, S. (2013). Middleware requirements for collaborative unmanned aerial vehicles. In *2013 International Conference on Unmanned Aircraft Systems, ICUAS 2013 - Conference Proceedings*, pages 1051–1060.

[107] Monniaux, D. (2005). Compositional analysis of floating-point linear numerical filters. In *Proceedings of the 17th International Conference on Computer Aided Verification*, CAV'05, page 199–212, Berlin, Heidelberg. Springer-Verlag.

[108] Morse, J., Cordeiro, L. C., Nicole, D., and Fischer, B. (2015). Model checking LTL properties over ANSI-C programs with bounded traces. *Software and System Modeling*, 14(1):65–81.

[109] Munea, T. L., Lim, H., and Shon, T. (2016). Network protocol fuzz testing for information systems and applications: a survey and taxonomy. *Multimedia Tools and Applications*, 75(22):14745–14757.

[110] Nassi, B., Ben-Netanel, R., Shamir, A., and Elovici, Y. (2018). Game of Drones-Detecting Streamed POI from Encrypted FPV Channel. In *MobiSys*.

[111] Nobles, C. (2017). Cyber Threats in Civil Aviation. In *Security Solutions for Hyperconnectivity and the Internet of Things*, pages 272–301. IGI Global.

[112] Pak, B. S. (2012). Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution.

[113] ParisTech (2013a). PIPS: Automatic parallelizer and code transformation framework. https://pips4u.org/. [Online; accessed August-2019].

[114] ParisTech, M. (2013b). PIPS: Automatic Parallelizer and Code Transformation Framework. Available at http://pips4u.org.

[115] Patelli, A. and Mottola, L. (2016). Model-based real-time testing of drone autopilots. In *Proceedings of the 2Nd Workshop on Micro Aerial Vehicle Networks, Systems, and Applications for Civilian Use*, DroNet '16, pages 11–16, New York, NY, USA. ACM.

[116] Pecheur, C. (2000). *Verification and validation of autonomy software at NASA*. NASA.

[117] Pleban, J.-S., Band, R., and Creutzburg, R. (2014). Hacking and securing the AR.Drone 2.0 quadcopter: investigations for improving the security of a toy. In Creutzburg, R. and Akopian, D., editors, *Mobile Devices and Multimedia: Enabling Technologies, Algorithms, and Applications 2014*, volume 9030, pages 168 – 179. International Society for Optics and Photonics, SPIE.

[118] Prasad, M. R., Biere, A., and Gupta, A. (2005). A survey of recent advances in SAT-based formal verification. *Software Tools for Technology Transfer*, 7(2):156–173.

[119] Psiaki, M. L., O'Hanlon, B. W., Bhatti, J. A., Shepard, D. P., and Humphreys, T. E. (2013). Gps spoofing detection via dual-receiver correlation of military signals. *IEEE Transactions on Aerospace and Electronic Systems*, 49(4):2250–2267.

[120] PwC (2018). Drones could add £42bn to UK GDP by 2030 - PwC research.

[121] PwC. (2018). Gatwick airport drones disruption wasn't all for nothing, UK police insist.

[122] Rival, X. and Mauborgne, L. (2007). The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29:26.

[123] Rocha, H., Ismail, H., Cordeiro, L. C., and Barreto, R. S. (2015). Model checking embedded C software using k-induction and invariants. In *Brazilian Symposium on Computing Systems Engineering*, pages 90–95.

[124] Rocha, W., Rocha, H., Ismail, H., Cordeiro, L., and Fischer, B. (2017a). Depthk: A k-induction verifier based on invariant inference for c programs. In Legay, A. and Margaria, T., editors, *TACAS*, pages 360–364, Berlin, Heidelberg. Springer Berlin Heidelberg.

[125] Rocha, W., Rocha, H., Ismail, H., Cordeiro, L., and Fischer, B. (2017b). Depthk: A k-induction verifier based on invariant inference for c programs. In *TACAS, LNCS 10206*, pages 360–364.

[126] Rocha, W., Rocha, H., Ismail, H., Cordeiro, L. C., and Fischer, B. (2017c). Depthk: A k-induction verifier based on invariant inference for C programs - (competition contribution). In *Tools And Algorithms For The Construction And Analysis Of Systems*, volume 10206 of *LNCS*, pages 360–364.

[127] rodday, N. (2015). *EXPLORING SECURITY VULNERABILITIES OF UNMANNED AERIAL VEHICLES*. PhD thesis, University of Twente.

[128] Rouff, C., Hinchey, M., Truszkowski, W., and Rash, J. (2005). Verifying large numbers of cooperating adaptive agents. In *11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, volume 1, pages 391–397 Vol. 1.

[129] Rudinskas, D., Goraj, Z., and Stankūnas, J. (2009). Security analysis of uav radio communication system. *Aviation*, 13(4):116–121.

[130] Rudo, D., Zeng, D., et al. (2020). Consumer uav cybersecurity vulnerability assessment using fuzzing tests. *arXiv preprint arXiv:2008.03621*.

[131] Scott, J., Lee, L. H., Arends, J., and Moyer, B. (1998). Designing the low-power m*CORE architecture. In *Power Driven Microarchitecture Workshop*, pages 145–150.

[132] Sheeran, M., Singh, S., and Stålmarck, G. (2000). Checking Safety Properties Using Induction And A SAT-Solver. In *FMCAD*, pages 108–125.

[133] Si, X., Dai, H., Raghothaman, M., Naik, M., and Song, L. (2018). Learning loop invariants for program verification. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 7762–7773. Curran Associates Inc.

[134] Singhal, G., Bansod, B., and Mathew, L. (2018). Unmanned aerial vehicle classification, applications and challenges: A review.

[135] Sirigineedi, G., Tsourdos, A., Żbikowski, R., and White, B. A. (2010a). Modelling and Verification of Multiple UAV Mission Using SMV. *EPTCS*, 20:22–33.

[136] Sirigineedi, G., Tsourdos, A., Żbikowski, R., and White, B. A. (2010b). Modelling and verification of multiple uav mission using smv. *Electronic Proceedings in Theoretical Computer Science*, 20:22–33.

[137] SNU (2012). Real-time benchmarks. http://www.cprover.org/goto-cc/examples/snu.html. [Online; accessed August-2019].

[138] Srivastava, S. and Gulwani, S. (2009). Program verification using templates over predicate abstraction. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 223–234, New York, NY, USA. Association for Computing Machinery.

[139] Strohmeier, M., Lenders, V., and Martinovic, I. (2015). Intrusion detection for airborne communication using phy-layer information. In *DIMVA*, pages 67–77.

[140] Tarabay, J. (2018). Israel: Iranian drone we shot down was based on captured US drone - CNN.

[141] Van Rossum, G. and Drake Jr, F. L. (1995). *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands.

[142] Wright, A. and Felleisen, M. (1994). A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94.

[143] Xu, H. and Wang, P. (2016). Real-time reliability verification for uav flight control system supporting airworthiness certification. *PLOS ONE*, 11(12):1–21.

[144] Yaacoub, J.-P. and Salman, O. (2020). Security analysis of drones systems: Attacks, limitations, and recommendations. *Internet of Things*, page 100218.

[145] Yeh, C.-C., Lu, H.-L., Yeh, J.-J., and Huang, S.-K. (2017). Path Exploration Based on Monte Carlo Tree Search for Symbolic Execution. In *2017 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pages 33–37. IEEE.

[146] Zutshi, A., Sankaranarayanan, S., Deshmukh, J. V., Kapinski, J., and Jin, X. (2015). Falsification of safety properties for closed loop control systems. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, HSCC '15, page 299–300, New York, NY, USA. Association for Computing Machinery.