ENHANCING AUTOMATIC SOFTWARE TESTING FOR BROADER APPLICABILITY

by

Marko Dimjašević

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

May 2018

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Marko Dimjašević**

has been approved by the following supervisory committee members:

Zvonimir Rakamarić,	Chair(s)	**May 25, 2017**	Date Approved
Eric Eide,	Member	**May 25, 2017**	Date Approved
Dimitra Giannakopoulou,	Member	**May 25, 2017**	Date Approved
Ganesh Gopalakrishnan,	Member	**May 25, 2017**	Date Approved
John Regehr,	Member	**May 25, 2017**	Date Approved

by **Ross Whitaker**, Chair/Dean of

the Department/College/School of **Computing**

and by **David B. Kieda**, Dean of The Graduate School.

ABSTRACT

In computer science, functional software testing is a method of ensuring that software gives expected output on specific inputs. Software testing is conducted to ensure desired levels of quality in light of uncertainty resulting from the complexity of software. Most of today's software is written by people and software development is a creative activity. However, due to the complexity of computer systems and software development processes, this activity leads to a mismatch between the expected software functionality and the implemented one. If not addressed in a timely and proper manner, this mismatch can cause serious consequences to users of the software, such as security and privacy breaches, financial loss, and adversarial human health issues. Because of manual effort, software testing is costly. Software testing that is performed without human intervention is automatic software testing and it is one way of addressing the issue.

In this work, we build upon and extend several techniques for automatic software testing. The techniques do not require any guidance from the user. Goals that are achieved with the techniques are checking for yet unknown errors, automatically testing object-oriented software, and detecting malicious software. To meet these goals, we explored several techniques and related challenges: automatic test case generation, runtime verification, dynamic symbolic execution, and the type and size of test inputs for efficient detection of malicious software via machine learning.

Our work targets software written in the Java programming language, though the techniques are general and applicable to other languages. We performed an extensive evaluation on freely available Java software projects, a flight collision avoidance system, and thousands of applications for the Android operating system. Evaluation results show to what extent dynamic symbolic execution is applicable in testing object-oriented software, they show correctness of the flight system on millions of automatically customized and generated test cases, and they show that simple and relatively small inputs in random testing can lead to effective malicious software detection.

To Lucija, my love.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This part of the dissertation took the most time among all of the parts for it to be able to be written. It was a journey that took five years. While the acknowledgments deserve their own chapter, I will try to keep them short while trying not to forget to thank anyone that helped with making this dissertation happen.

I thank members of my dissertation committee, Eric Eide, Ganesh Gopalakrishnan, and John Regehr, for discussions and their guidance in getting to this point. Meeting them to talk about my progress and asking them for suggestions on what to do next defined directions of my actions. Furthermore, I am thankful to Falk Howar, Kasper Søe Luckow, Malte Isberner, Temesghen Kahsai, and Vishwanath Raman for the fruitful research collaborations we have had. With most of them, I wrote several papers that directly contributed to this dissertation. I have known Ivo Ugrina since my undergraduate days in Croatia. Even though I never thought the two of us would write a research paper together because his work is on statistics, we had a pleasant collaboration and wrote a paper during my doctorate, for which I am very thankful. Simone Atzeni has been a great friend throughout my PhD and helped in my personal life many times. It was also very rewarding to work with Simone on what started as a class project, but that soon grew into a research project. Thanks to the collaboration with him and Ivo, this dissertation includes work on malware detection presented in Chapter 2.

I got to know Raimondas Sasnauskas when he joined the School of Computing as a postdoc. He helped me a lot with the Android platform in one of the projects that is part of this dissertation. Raimondas has also been a great friend and a person with whom to discuss various topics, from computer science to politics and culture. The two of us are hoping to write a research paper together at some point. During my PhD, I made friends with several members of the software verification group at the university and I am glad to have known them: Mark Baranowski, Diego Caminha Barbosa de Oliveira, Geof Sawaya, Sriram Aananthakrishnan, Montgomery Carter, and Alan Humphrey. However, there are two members of the group whose friendship has been something I appreciate a lot: Shaobo He and Mohammed Saeed Al-Mahfoudh. With both of them, I spent a lot of time in sport activities, which we also used as an excuse to discuss numerous topics, which I am thankful for.

Thanks to Dimitra Giannakopoulou, this dissertation has Chapter 4, which is based on our joint work on automatic testing and runtime verification. Dimitra was my mentor during my internship at NASA and it was such a rewarding experience to work with her. Furthermore, our collaboration was fruitful in other projects, too, and work presented in Chapter 3 is based on a few more publications the two of us coauthored.

My thesis advisor Zvonimir Rakamarić, obviously, played a major role in this doctorate. Zvonimir was there to help from the very beginning when I was applying for the PhD program at the University of Utah. Since then, he provided guidance and expertise in everything related to research and academia, from my very first steps in reading about and dissecting other researchers' work to starting my own lines of research work. In these five years, we had so many interactions, meetings, and discussions that they are even hard to count. This led to this point where I am presenting in dissertation form all of the work that I have done in this time period. I am very thankful to Zvonimir for all of the things mentioned above and for showing what is it like to do research.

A technical infrastructure that defined my PhD is the Emulab testbed, which I have used extensively throughout these five years. The infrastructure has been developed and maintained by the Flux Research Group of the University of Utah and I am thankful to all members of the group — most of all to Mike Hibler for the countless times he replied to my emails and provided help — for their quick responses and help with troubleshooting problems I had with the infrastructure and for being generous in providing additional hardware resources. I always knew I could count on them to provide a reliable infrastructure for my research.

The work I have done toward this dissertation was supported by several sources. It was supported in part by the National Science Foundation's Division of Computing and Communication Foundations Grant No. 1421678, by the United States National Aeronautics and Space Administration (NASA) Ames Research Center, by NASA under Prime Contract No. NNA10DE60C, by the International Alumni Club of the University of Utah, and by the Google Summer of Code 2013 program.

Even though temporarily we ended up in distant parts of the world when I moved from Croatia to the United States for my PhD, friendship with the following friends that I have known for a long time has not faded: Marko Matosović, Ivan Soković, Miroslava Harča, Marko Šoštarić, and Goran Ljaljić. We have stayed in touch all the time and spent time together in person on a few occasions when I visited Croatia. Furthermore, two more long-time friends, Ivan Radiček and Andrej Dundović, have also been in Europe during this time, although they moved from Croatia to other European countries in pursuit of their doctoral degrees. They too have been important in my personal life, but also in my academic life by sharing their experiences in doing science in western Europe.

This dissertation would not have been possible if it were not for my family. With the help of communication technologies, we have been in touch all the time. Thank you mom Anica, Marija, Ivana, Snježana, Ivica, Ruža, Ema, Leonardo, Mario, Helena, David, Lana, Danijel, Dora, Luka, Karlo, Petra, and Ana for countless messages, calls, jokes, and nice words and moments! You made it so much easier to go through this challenging five-year journey. I missed you all and I am looking forward to moving back to Croatia so that I can be at family gatherings in person again, and not only virtually.

I met the love of my life during my PhD. She, along with the rest of my family, has provided emotional support and encouragement when I needed it most. I am so lucky to have Lucija, my fiancée, in my life! She has been making a big sacrifice in waiting for me to return to Croatia and I am relieved this waiting is coming to an end.

In Salt Lake City, May 2017

CHAPTER 1

INTRODUCTION

Since the late twentieth century, we have been witnessing a shift in how society functions by relying more and more on computing devices. Even though they come in different form factors and serve different purposes, what is common to all computing devices is that they are controlled by software. In other words, computing devices are programmed via software to do certain tasks. Today's software accomplishes complex tasks such as flying an aircraft. Other tasks include telecommunication operation, navigation, concert ticket reservation, email client, and calendar organizer. Without doubt, software has enhanced people's everyday lives and work.

Software has to be written and consequently maintained. These two challenging and creative activities are commonly referred to as software development, which is performed by people. Software development is carried out by both individuals and small and large groups, facilitated by existing software tools. To deal with complexity of software, software developers organize software that they are working on in layers of increasing abstraction. The lowest layer communicates with hardware and provides services to upper layers by abstracting away hardware specifics. The highest layers provide users with services such as video calls. In spite of such organization of software, each layer is still rather complex. Decades of software development have shown that the complexity of software inevitably leads to it malfunctioning, which can have serious consequences to its users, including security and privacy breaches, financial loss, and adverse human health issues.

Because developing software with no unwanted behavior is extremely hard, developers try to minimize the number of such behaviors and their impact. One viable approach to addressing this quality challenge is software verification. It consists of writing a formal specification of what the target software should do and then proving that the implementation of the target software corresponds to the specification. Proving even a simple specification typically requires much more proof theory knowledge and time than software developers have, hence this approach is rarely taken.

Another approach is software testing. It is a common approach to finding errors and unwanted behavior in software and making sure no regressions are introduced. It consists of providing software with known-in-advance input and checking if it produces expected output. When an input is found that makes software produce an unexpected output, i.e., an error occurs, the software is fixed and the process is repeated. The ideal goal is to cover all possible inputs to make sure no errors are possible. However, for any nontrivial software there are far too many inputs to cover in a limited amount of time. Therefore, only some inputs are tested for.

Software testing is done either completely manually or semi-automatically by having a domain expert guide a testing tool. When done manually, testers take the role of end users and use software trying to discover errors, while developers write test cases that cover particular inputs. In a semi-automatic case, a domain expert specifies what and how to test such that a testing software tool can perform tasks that are otherwise done manually. In both cases, software testing turns out to be overwhelmingly time-consuming. Therefore, there is a need for automatic testing.

There are benefits to testing in an automatic fashion: 1) it reduces time needed for testing, 2) it can be replayed on different variants of the same software, and 3) it reduces human error that can happen in testing. Towards automating the testing process, developers write auxiliary software that tests target software. Such auxiliary software presents the target software with specific inputs that developers have thought of and decided to test it on. Nevertheless, that leaves a lot of inputs uncovered, i.e., it leaves a lot of room for inputs that can make the target software behave unexpectedly. As illustrated earlier, software plays increasingly more important roles in our lives, yet dominant practices in ensuring desired software quality are time-consuming and suboptimal.

An extended solution to addressing this quality challenge is to perform automatic software testing. There are two parts to automatic testing: 1) generating inputs, and 2) checking if desired properties hold for generated inputs. To utilize automatic testing, developers have to formulate properties that software under test should have, encode them in a software implementation, think of domains over which inputs to be generated should range, and implement generators for inputs. Such testing can reveal misconceptions about desired software behavior as well as undesired behavior like errors and security threats that are triggered on inputs that would otherwise be left untested.

In this dissertation, we consider several techniques for automatic software testing and broaden their applicability. They address the challenge of identifying previously unknown errors and security threats in software. The techniques are automatic, i.e., they do not require interaction with the user. In addressing different problems, we investigate several techniques and related challenges: automatic test case generation, runtime verification, dynamic symbolic execution, and the type and size of test inputs for efficient detection of malicious software via machine learning. All of these techniques increase software reliability and security.

One of the main goals of this work is to make the techniques work at the scale of real-world software, without impeding the usability of the techniques or having to substantially instrument the software under analysis. This goal is important if the techniques are to be applicable to such complex software. All three lines of work required no modification of the software under analysis. This enables developers of the software to retain their usual development processes while also providing the benefit of automatically finding errors in the software.

The scalability goal in some of the techniques was achieved by combining existing methods in novel ways. In one line of work, we used random testing to dynamically analyze software executions. For each application considered, we chose a parameterized number of sampled inputs that served as application execution drivers. These inputs mimicked user interaction with the application. While each application was executing, we traced its system calls that occurred due to the provided inputs. Then we repeated this for thousands of applications, processed each application's system calls, and used that in machine learning to gain an insight into what makes an application malicious. Therefore, we used automatic testing to observe application maliciousness that was automatically modeled via machine learning techniques.

Similarly, automatic testing of object-oriented software was accomplished by interleaving feedback-directed random testing and dynamic symbolic execution. One of the main challenges in automatically testing object-oriented software is construction of appropriate objects in the heap memory, which would serve as input to the rest of the program. When interleaved, the two techniques enabled us to deal with the sheer number of different executions that dynamic symbolic execution would have to explore on its own otherwise. With feedback-directed random testing, we first construct multiple execution paths that created objects and invoked methods on them. Then we applied dynamic symbolic execution to explore paths that branch off of the constructed paths. Therefore, the interleaving approach provided us with an ability to sample from inputs and explore execution paths that they can result in.

In the line of work on verifying at runtime an aircraft separation assurance system, we verified properties by monitoring test cases during their execution. There we used a novel approach in generating inputs of interest. Inputs in this line are aircraft trajectories, where each trajectory is a sequence of points in space and time. It is a four-dimensional space, where three dimensions are an aircraft's latitude, longitude, and height position, while the remaining one is time when an aircraft was at that position. The main challenge there is to create a property-covering scenario with multiple in-flight aircraft and their trajectories. Because simply sampling from the input space of multiple trajectories is extremely likely to give an uninteresting input on which to test and verify properties, an approach that would find interesting inputs was needed. Our solution was to create somewhat interesting trajectories first, provide them as input to the system, monitor and learn from executions, and then create more interesting and complex test cases at runtime.

This dissertation is organized around three lines of work. In the first part of the dissertation, we analyze how the type and size of inputs in random testing affects malware detection abilities for a widely used mobile computing operating system. In the second part, we look at the problem of automatically testing object-oriented software. Our work addresses the problem by combining feedback-directed random testing and dynamic symbolic execution. The last part is on runtime verification of a complex real-world software system. There we consider how runtime monitoring can be interleaved with automatic test case generation in order to target property coverage. With that said, our thesis statement is the following:

> Automatic software testing can be combined with machine learning, dynamic symbolic execution, and runtime verification to broaden its applicability.

We implemented the techniques for software written in the Java programming language. The benchmarks on which we evaluated our techniques include dozens of freely available Java software projects, a flight collision avoidance system, and thousands of applications for the Android operating system. Such a wide spectrum of benchmarks and their size and complexity demonstrate the broad applicability of the proposed automatic software testing techniques. The results of our work show that: 1) simple inputs in random testing can be used to effectively detect malicious software, 2) for object-oriented software, dynamic symbolic execution provides additional code coverage on top of feedback-directed random testing, and 3) the correctness of a flight system can efficiently be tested on millions of automatically customized and generated test cases.

CHAPTER 2

AUTOMATIC TESTING FOR MALWARE DETECTION

This chapter is based on work published at the International Workshop on Security And Privacy Analytics 2016 [DAUR16a] and on the accompanying technical report [DAUR15].[1]

2.1 Introduction

The global market for mobile devices has exploded in the past several years, and according to some estimates, the number of smartphone users alone reached 1.7 billion worldwide in 2014. Android is the most popular mobile platform, holding nearly 85% of the global smartphone market share. One of the main advantages of mobile devices such as smartphones is that they allow for numerous customizations and extensions through installing applications from public application repositories. The largest of such repositories (e.g., Google Play and the Apple App Store) have more than one million applications available for download each, and there are more than 100 billion mobile device applications installed worldwide.

This clearly provides a fertile environment for malicious activities, including the development and distribution of malware. A recent study [jun13] estimates that the total amount of malware across all mobile platforms grew exponentially at the rate of 600% between 03/2012 and 03/2013. Around 92% of the malware applications found in this study target Android. In a related study [ris14], similar statistics are reported — the number of malicious applications in the Google Play store grew around 400% from 2011 to 2013, while at the same time, the percentage of malicious applications removed annually by Google has dropped from 60% in 2011 to 23% in 2013. Due to the sharp increase in the total amount of malware, the percentage of removed malware dropped significantly despite the fact that the absolute number actually increased from roughly 7,000 in 2011 to nearly

[1]Portions of the published work are reused and reprinted here with permission.

10,000 in 2013. Alcatel-Lucent estimates the mobile malware infection rate to be around 0.65%, which means that around 15 million mobile devices are infected with malware, 60% of which run Android [alc14]. A recent research paper found that the malware infection rates in Android devices are 0.28% and 0.26%, depending on the chosen malware data set [TLN$^+$14]. While companies such as Google regularly scan their application repositories using proprietary tools, this process is often ineffective as the above numbers illustrate. There are also unofficial, open repositories where often no scanning is performed, partially because there is a lack of solid freely available solutions and tools. As a consequence, Android malware detection has been an active area of research in the past several years, both in industry and academia.

Currently, published approaches can be broadly categorized into manual expert-based approaches, and automatic static- or dynamic-analysis-based techniques. Expert-based approaches detect malware by relying on manually specified malware features, such as requested permissions [ADY13] or application signatures [GZZ$^+$12, FADA14]. These require significant manual effort by an expert user, are often easy to circumvent by malware writers, and target existing, specific types of malware, thereby not providing protection from evolving malicious applications.

Static-analysis-based techniques typically search for similarities to known malware. This often works well in practice since new malware samples are typically just variations of existing ones. Several such techniques look for code variations [CGC12, HHW$^+$13], which becomes ineffective when faced with advanced code obfuscation techniques. Hence, researchers have been exploring more high-level properties of code that can be extracted statically, such as call graphs [GYAR13], which make these techniques more resilient to code obfuscation. Unfortunately, even those approaches can be evaded by leveraging well-known drawbacks of static analysis. For example, generated call graphs are typically over-approximations, and hence can be obfuscated by adding many dummy, unreachable function calls. In addition, native code is hard to analyze statically, and hence malicious behavior can be hidden there.

Dynamic analysis techniques typically run applications in a sandbox environment or on real devices in order to extract information about the application behavior. The extracted information is then automatically analyzed for malicious behavior using various techniques, such as machine learning. Recent techniques is this category often observe application behavior by tracing system calls in a virtualized environment [BBS+10, RFC13, LBK+10]. Both static analysis and dynamic analysis proponents made various claims, often contradicting ones — including claims that are based on questionably designed experiments — on effectiveness of malware detection based on system calls.

In this chapter, we evaluate existing and propose novel dynamic Android malware detection techniques based on automatic testing and on tracking of system calls, all of which we implemented as a free software tool called MALINE. Our work was initially inspired by a similar approach proposed for desktop malware detection [PBCK13], albeit we provide simpler feature encodings and an Android-specific tool flow. We provide several encodings of behavior fingerprints of applications into features for subsequent classification. We performed an extensive empirical evaluation on a set of more than 12,000 Android applications. We analyze how the quality of malware classifiers is affected across several dimensions, including the choice of an encoding of system calls into features, the relative sizes of benign and malicious data sets used in experiments, the choice of a classification algorithm, and the size and type of inputs in automatic testing that drive a dynamic analysis. Furthermore, we show that the structure of system call sequences observed during application executions conveys in itself a lot of information about application behaviors. Our evaluation sheds light on several such aspects, and shows that the proposed combinations can be effective: our technique yields an overall detection accuracy of 93% with a 5% benign application classification error. Finally, we provide guidelines for domain experts when making choices on malware detection tools for Android, such as MALINE.

Our approach provides several key benefits. By guarding the users at the repository level, a malicious application is detected early and before it is made publicly available for installation. This saves scarce energy resources on the devices by delegating the detection task to a trusted remote party, while at the same time protecting users' data, privacy, and payment accounts. System call monitoring is out of reach of malicious applications, i.e., they cannot affect the monitoring process. Hence, our analysis that relies on monitoring system calls happens with higher privileges than those of malicious applications. In addition, tracking system calls entering the kernel (and not calls at the Java library level) enables us to capture malicious behavior potentially hidden in native code. Since our approach is based on coupling an automatic testing-guided dynamic analysis with classification based on machine learning, it is completely automatic. We require no source code, and we capture dynamic behavior of applications as opposed to their code properties such as call graphs; hence, our approach is mostly immune to common, simple obfuscation techniques. The advantages of our approach make it complementary to many existing approaches, such as the ones based on static analysis.

Our contributions are summarized as follows:

- We show that automatic testing can effectively be applied to as disparate an area as malware detection for mobile platforms.

- We propose a completely automatic approach to Android malware detection at the application repository level using automatic testing, system call tracking, and classification based on machine learning, including a novel heuristics-based encoding of sequences of system calls into features.

- We implement the approach in a tool called MALINE, and perform extensive empirical evaluation on more than 12,000 applications. We show that MALINE effectively discovers malware with a very low rate of false positives.

- We compare several feature extraction strategies and classifiers. In particular, we show that the effectiveness of even very simplistic feature choices (e.g., the frequency of system calls) is comparable to much more heavyweight approaches. Hence, our results provide a solid baseline and guidance for future research in this area.

- Finally, we contribute 300 GB of data [DAUR16b] generated during this work, for other researchers, teachers, and the general public to inspect, use, and build upon in their work. For example, the data were already used in a teaching setting in a class at the University of Utah where students built machine learning algorithms, which they compared using our data.

2.2 Preliminaries

In this section, we introduce a problem definition and preliminaries for the problem.

2.2.1 Problem Definition

We are provided with a set of Android applications prelabeled as either benign (goodware) or malicious (malware). In addition, we assume that each application can be classified as benign or malicious based on its behavior, i.e., the actions it performs. The goal is to learn behavioral characteristics of applications to be able to discriminate benign from malicious ones in a new, yet unlabeled set.

2.2.2 System Calls

A system call is a mechanism for a program to request a service from the underlying operating system's kernel. In Android, system calls are created by information flowing through a multilayered architecture depicted in Figure 2.1. For example, an Android text messaging application, located at the highest level of the architecture, might receive a user request to send an SMS message. The request is transformed into a request to the Telephony Manager service in the Application Framework. Next, the Android runtime receives the request from the service, and it executes it in the Dalvik Virtual Machine.[2] The execution transforms it into a collection of library calls, which eventually result in multiple system calls being made to the Linux kernel. One of the system calls will be to `sendmsg`:

```
sendmsg(int sockfd, const struct msghdr* msg, unsigned int flags)
```

The `sendmsg` function is used to send a message on a socket. The generated sequence of system calls is a low-level equivalent of the SMS message being sent in the application at the highest level of abstraction. Information flows in the opposite direction in a similar fashion.

2.2.3 Automatic Testing

In our approach to malware detection, we heavily rely on automatic testing. We use a form of random automatic testing where pseudo-random events are generated and provided as inputs to an Android application or the Android system. Android applications are event-driven; hence, the generated inputs drive the execution of applications. In this way, the state of an application in its execution is changed based on the provided input. A sequence

[2]As of Android version 5.0, the Dalvik Virtual Machine was replaced with an application runtime environment called ART.

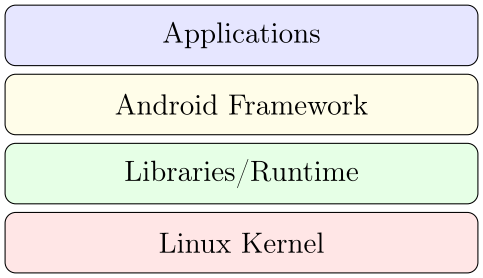

Figure 2.1. Abstraction layers of the Android architecture.

of inputs will cause the application to go through a sequence of states. Consequently, this reflects in a corresponding sequence of system calls that happen between the application and the Android operating system.

In the testing phase of the malware detection approach, we vary the size and type of generated inputs, which causes an application to go through different states in its execution. By varying the inputs, we can drive an application's execution into different end states. This lets us observe at the system call level how the behavior of the application changes.

The Android Software Development Kit contains an automatic testing tool called Monkey [mon17]. Monkey generates pseudo-random events such as gestures, clicks, touches, as well as system-level events. We leverage Monkey in the dynamic phase of our approach to generate input events that drive the execution of an application.

2.2.4 Machine Learning

Our malware detection problem is an instance of a classification problem in machine learning, and is solved using a classifier algorithm. More specifically, it is an example of a binary classification problem since it explores connections between the behavior of an application and its goodware/malware label. The two groups are commonly called a positive and a negative group. If a positive example (e.g., an application in our case) is classified into the positive (i.e., respectively, negative) group, we obtained a true positive/TP (i.e., respectively, false negative/FN). Analogously, we define true negative/TN and false positive/FP. Table 2.1 gives standard measures of the quality of classification prediction used in machine learning based on these terms.

Classification is usually conducted through individual measurable properties of a phenomenon being investigated (e.g., the heights of people, their weights, or the number of

Table 2.1. Standard classifier quality measures. P (respectively, N) is the number of positive (respectively, negative) examples.

Measure	**Formula**
accuracy, recognition rate	$\frac{TP+TN}{P+N}$
errorrate, misclassification rate	$\frac{FP+FN}{P+N}$
sensitivity, true positive rate, recall	$\frac{TP}{P}$
specificity, true negative rate	$\frac{TN}{N}$
precision	$\frac{TP}{TP+FP}$

system calls in one run of an Android application). Such properties are called features, and a set of features of a given object is often represented as a feature vector. Feature vectors are stored in a feature matrix, where every row represents one feature vector.

Cross-validation is a technique for estimating the performance of a predictive model. If the validation is five-fold, it means that the input data are split into five disjoint subsets, where each subset in turn is used as a testing subset. Finally, double five-fold cross-validation means that the validation was performed for two parameters.

More about machine and statistical learning can be found in related literature [HTF09, JWH13].

2.3 Our Approach

Our approach is a three-phase analysis, as illustrated in Figure 2.2. The first phase is a dynamic analysis where we track system calls[3] during the execution of an application in a sandbox environment and record them into a log file. In the second phase, we encode the generated log files into feature vectors according to several representations we define. The last phase takes the feature vectors and applies machine learning [HTF09] to learn to discriminate benign from malicious applications. In both the second and third phase, we look at multiple techniques, as explained in the remainder of this section.

[3]A system call is a mechanism for a program to request a service from the underlying operating system's kernel. In Android, system calls are created by information flowing through its multilayered architecture, starting from an application on top; hence, they capture its behavior.

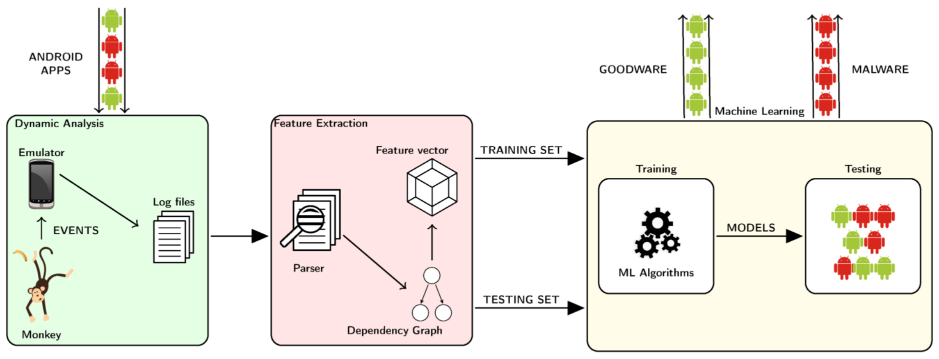

Figure 2.2. MALINE tool flow divided into three phases.

2.3.1 Dynamic Analysis Phase

As our approach is based on concrete executions of applications, the first phase tracks and logs events at the operating system level that an application causes while being executed in a sandbox environment. The generated event logs serve as a basis for the subsequent phases of our analysis. Unlike numerous static analysis techniques, this approach reasons only about events pertaining to the application that are actually observed in the operating system.

The dynamic analysis phase starts with random automatic testing for which we utilize the Monkey tool [mon17]. We rely on automatic testing to drive an application execution as a proxy to a person using an application. As in the case of a person using an application, automatic testing will result in system calls that characterize the interaction of the application with the Android system.

We execute every application in a sandbox environment and observe the resulting system calls in a chronological order, from the beginning until the end of its usage. The output of this phase is a log file containing chronologically ordered system calls: every line consists of a time stamp, the name of the system call, its input values, and the return value, if any. Having the system calls recorded chronologically enables us to construct various feature vectors that characterize the application's behavior with different levels of precision, as explained in the next section.

More formally, let $\mathcal{S} = \{s_1, s_2, \dots, s_n\}$ be a set of system call names containing all the system calls available in the Android operating system for a given processor architecture. Then a system call sequence σ of length m, representing the chronological sequence of recorded system calls in a log file, is a sequence of instances of system calls $\sigma = (q_1, q_2, \dots, q_m)$, where $q_i \in \mathcal{S}$ is the ith observed system call in the log file. Such call sequences are passed to the feature extraction phase.

2.3.2 Feature Extraction Phase

As explained earlier, how features are picked for the feature vector is important for the machine learning classification task. Therefore, we consider two representations for generating a feature vector from a system call sequence σ. Our simpler representation is concerned with how often a system call happens, while our richer representation encodes information about dependencies between system calls. Both representations ignore system call information other than their names and sequence numbers (e.g., invocation time, input and output values), as can be seen from the definition of σ. Once we compute a feature vector $\mathbf{x}$ for every application under analysis according to a chosen representation, we form a feature matrix by joining the feature vectors such that every row of the matrix corresponds to one feature vector.

2.3.2.1 System Call Frequency Representation

How often a system call occurs during an execution of an application carries information about its behavior [BZNT11]. One class of applications might be using a particular system call more frequently than another class. For example, some applications might be making considerably more I/O operation system calls than known goodware, indicating that the increased use of I/O system calls might be a sign of malicious behavior. Our simple system call frequency representation tries to capture such features. In this representation, every feature in a feature vector represents the number of occurrences of a system call during an execution of an application. Given a sequence σ, we define a feature vector $\mathbf{x} = [x_1 x_2 \dots x_{|\mathcal{S}|}]$, where x_i is equal to the frequency (i.e., the number of occurrences) of system call s_i in σ. In the experiments in Section 2.5, we use the system call frequency representation as a baseline comparison against the richer representation described next.

2.3.2.2 System Call Dependency Representation

Our system call dependency representation was inspired by previous work that has shown that a program's behavior can be characterized by dependencies formed through information flow between system calls [FJC+10]. However, we have not been able to find a tool for Android that would provide us with this information and also scale to analyzing thousands of applications. Hence, we propose a novel scalable representation that attempts to capture such dependencies by employing heuristics. As we show in Section 2.5, even though our representation is simpler than the one based on graph mining and concept analysis from the original work [FJC+10], it still produces feature vectors that result in highly accurate malware detection classifiers.

For a pair of system calls q_i and q_j in a sequence σ, where $i < j$, we define the distance between the calls as $d(q_i, q_j) = j - i$. We then approximate a potential data flow relationship between a pair of system calls using the distance between the calls in a sequence (i.e., log file). For example, if two system calls are adjacent in σ, their distance will be 1. Furthermore, let $w_{g,h}$ denote the weight of a directed edge (s_g, s_h) in a system call dependency graph we generate. The system call dependency graph is a complete digraph with the set of vertices being the set of all the system call names $\mathcal{S}$, and hence having $|\mathcal{S}|^2$ edges. Then, $w_{g,h}$ for a sequence σ is computed as:

$$w_{g,h} = \begin{cases} 0, & \text{if } \; g = h \\ \sum\limits_{\substack{i<j<k, \\ q_i = s_g, q_j = s_h}} \frac{1}{d(q_i, q_j)}, & \text{otherwise} \end{cases}$$

where k is the minimal index such that $q_i = q_k$ and $i < k \leq |\sigma|$. Informally, the closer the pair is in a sequence, the more it contributes to its edge weight in the graph. Hence, instead of explicitly observing a data flow between system calls, our representation captures it implicitly: it is based on a simple observation that the closer a pair of system calls is in a sequence, the more likely it is that there is a data flow between the pair.

From a sequence σ, we compute weights $w_{g,h}$ for every system call pair $(s_g, s_h) \in \mathcal{S}^2$. For g and h such that $w_{g,h} = 0$, we still consider edge (s_g, s_h) to exist, but with a weight of 0. Since each application is executed only once during our dynamic analysis phase, we generate one system call dependency graph per application.

We generate a feature vector $\mathbf{x}$ of an application by taking edge weights from its system call dependency graph. For every directed edge (s_g, s_h), there is a corresponding feature in $\mathbf{x}$, and hence the dimensionality of $\mathbf{x}$ is $|\mathcal{S}|^2$. Given a sequence σ, we define a feature vector $\mathbf{x} = [x_1 x_2 \ldots x_{|\mathcal{S}|^2}]$, where x_i is equal to $w_{g,h}$ such that $i = (g-1) \cdot |\mathcal{S}| + h$. Informally, $\mathbf{x}$ is a linearization of the array corresponding to the system call dependency graph.

2.3.3 Machine Learning Phase

We use the generated feature vectors for our applications (i.e., feature matrices) together with provided malware/goodware labels to build classifiers. We experimented with several of the most popular and effective classifiers: support vector machines (SVMs), random forest (RF), LASSO, and ridge regularization [HTF09]. We used the double cross-validation approach to tune parameters of classifiers.

When a probabilistic classifier is used, a threshold that appropriately tunes the trade-off between sensitivity and specificity can be studied using receiver operating characteristic (ROC) curves [HTF09]. Generating ROC curves is especially valuable to the users of malware detectors such as ours, since they can use them to fine-tune sensitivity vs. specificity depending on the intended usage. Hence, we generate ROC curves for the most interesting classifiers.

In our data set, around 33% samples are malware and the rest are goodware. Although this approach does not generate a perfectly balanced design, it tries to represent the goodware population in the best possible manner while still keeping a high percentage of malware samples and keeping computational costs at a practical level. In addition, we explored what can be achieved by balancing the design through resampling strategies of up-sampling (or over-sampling) the minority class and down-sampling (or under-sampling) the majority class [KJ13] implemented through bootstrapping.

2.4 Implementation

We implemented our approach in a tool called MALINE, and Figure 2.2 shows its tool flow. The implementation comes as a free and open reproducible research environment to foster further evaluation, development, and research in this area.[4] Our experience working on this project suggests that there is a lack of open, stable, and extensible infrastructures for performing dynamic security analysis of Android. Hence, we have invested significant effort into developing MALINE to be an extensive and reproducible research infrastructure

[4]The MALINE tool is available from `https://github.com/soarlab/maline` under the GNU Affero GPLv3 license.

enabling execution of easy-to-repeat experiments in the wider domain of Android security. MALINE heavily utilizes our own build of the Android Software Development Kit (SDK) and in Section 2.4.1.1, we discuss specifics we introduced to the SDK. The SDK includes the Android Emulator, which runs a virtual machine (VM) with the Android operating system. Every application MALINE analyzes is installed, executed, and monitored in the VM. The tool primarily resides on the host machine and relies on the Android Debug Bridge (ADB) to communicate with the VM. The bridge is used, for example, to push and install an application from the host machine into the VM, to check the VM's status, and to pull a log file from the VM to the host machine.

Subsequently, MALINE generates feature vectors on the host machine and feeds them to several machine learning libraries we used.

2.4.1 Host and Emulator

MALINE consists of a number of smaller components. We implemented multiple interfaces on the host side, ranging from starting and monitoring an experiment with multiple emulator instances running in parallel to machine-learning differences between applications based on the processed data obtained from emulator instances. It is the host side that coordinates and controls all such activities. For example, it creates and starts a pristine installation of Android in an emulator instance, then installs an application in it, starts the application, and waits for the application to finish so it can analyze system calls the application has made during its execution.

We use the emulator, which is built on top of QEMU [Bel05], in the dynamic analysis phase of our approach (see Figure 2.2). For every application, we create a pristine sandboxed environment since the emulator enables us to easily create a clean installation of Android. It is important that each application is executed in a clean and controlled environment to make sure nothing is left behind from previous executions and to be able to monitor the execution. Hence, every application's execution is completely independent of executions of all the other analyzed applications.

2.4.1.1 Custom Build of Android SDK

In our implementation, we used the Android 4.4.3 KitKat release, which utilizes Android API version 19. However, we have our own build of the Android system implemented on top of the official source code repositories. The main reason for the custom build is to avoid bugs we found in the Android SDK throughout multiple releases.

For example, one release had a bug in the Android emulator, making it unable to boot a virtual machine snapshot. Another release had a problem with the emulator sometimes becoming unresponsive for hours.

Our build features a modification to the Monkey tool (we describe the tool later) to have better control over experiments. The default Monkey version injects an event into a system queue and moves onto the next event right away, without waiting for the queued event to be executed. However, to make Android more responsive, its developers decided to drop events from the queue when under heavy load. In our experiments, this would mean that events that Monkey injects might be discarded, thereby compromising the dynamic analysis of an application under test. To make sure the Android system does not drop events, we have slightly modified Monkey so that it waits for each event to be executed before proceeding to the next event. We made this custom build freely available, and we reference it in the MALINE documentation.

2.4.2 Automatic Testing of Applications

In order to scale to thousands of applications, our dynamic analysis phase implements an automatic application testing and execution process. The process starts with making a clean copy of our default VM. The copy contains only what is installed by default in a fresh installation of the Android operating system from the Android Open Source Project. Once the installation boots, we use ADB to send an application from the host machine to the VM for installation. Next, we start the application and immediately begin tracing system calls related to the operating system process of the application with the strace tool. The system calls are recorded into a log file.

We simulate a user interaction with an Android device by injecting both internal and external events into the emulator. Internal events are sent to the application itself, such as screen clicks, touches, and gestures. We use the Monkey tool [mon17] as our internal event generator (see Figure 2.2). It sends a parameterized number of the events to the application, with a 100 ms pause period between consecutive events if applicable.[5] Unlike

[5]The pause between two consecutive events may not be applicable to actions that are time-dependent,

internal events, which are delivered to the application, external events are delivered to the emulator and include events that come from interacting with an external environment. In our experiments, for external events, we focus on generating text messages and location updates only since those are sometimes related to malicious behaviors.

We stop an application execution when all internal events generated by Monkey are delivered and executed, and then we pull the log file from the VM to the host machine for parsing. Next, we apply a feature vector representation, either the system call frequency or dependency representation as explained in Section 2.3. The output is a textual feature vector file per log file, i.e., per application, listing all the features. Finally, we combine all the feature vectors into a single matrix where each matrix row corresponds to one feature vector, i.e., one application.

2.4.3 Classification

Using the feature matrix generated from logs and previously obtained labels denoting malware/goodware for applications, we proceed with classification. As mentioned in Section 2.3.3, we experimented with several classification algorithms: random forest, SVMs, LASSO, and ridge regression. An implementation of SVMs is based on libSVM [CL11], while all the other algorithms are implemented in R [r17] using the language's libraries. The scripts are heavily parallelized and adjusted to be run on large machines or clusters. For example, running a random forest model on a feature matrix from a system call dependency graph sample requires 32 GB of RAM in one instance of five-fold cross-validation.

2.5 Experimental Evaluation

We evaluated MALINE using a set of 32-core machines with 128 GB of RAM running Ubuntu 12.04. The machines are part of the Emulab infrastructure [WLS$^+$02]. Our Android virtual machines running on the computer had full Internet access during experiments. This network configuration enabled applications under analysis an almost unimpeded access to whatever websites or resources they needed to access during their execution. Emulab does, however, employ a firewall blocking most low numbered ports, which prevents a malicious

such as screen tapping. Furthermore, the pause is not to be confused with our modification to the SDK regarding queued event execution.

application from becoming a source of a DoS attack to sites outside of Emulab [emu15]. We wrote scripts to automatize and parallelize our experiments, without which our extensive experimental evaluation would not be possible. In our experiments, we use only the x86 Android emulator; the resulting x86 system call set $\mathcal{S}$ has 360 system calls.

As explained in Section 2.4.1.1, we have our own custom build of the Android SDK version 4.4.3. Because the Android Open Source Project depends on more than four hundred Git repositories, we needed to track the exact versions of the repositories used and the changes we make on top of them. In the MALINE documentation, we record the exact versions of each repository used in the custom build of the SDK.

2.5.1 Input Data Set

We obtained applications from Google Play as goodware. Our malware applications are from the Drebin data set [ASH+14]. Before we could start using the collected applications in MALINE, we performed a filtering step. First, we removed applications that we failed to consistently install in the Android emulator. For example, even though every Android application is supposed to be self-contained, some applications had dependencies that were not installed at the time; we do not include such applications in our final data set. Second, we removed all applications that we could not consistently start or that would crash immediately. For example, unlike typical Android applications, application widgets are miniature application views that do not have an Android Activity, and hence they cannot be started from a launch menu. Third, with some applications, one of the first two reasons was observed only in some experiment setups. In order to have consistent data sets across all experiments, we filter out such applications as well.

Applications in the Drebin data set were collected between August 2010 and October 2012, and filtered by their collectors to contain only malicious applications. To the best of our knowledge, this is the latest verified malware application collection of its size used by researchers. The malicious applications come from more than 20 malware families and are classified based on how an application is installed and activated, or based on its malicious payloads [ZJ12]. The aim of our work is not to explore the specifics of the families; many other researchers have done that. Therefore, in our experiments, we make no distinction between malicious applications coming from different families. Rather, our focus is on: 1) leveraging random automatic testing to generate input events for an application, 2) analyzing how stimulation in terms of events sent to an application and to the emulator affects the overall ability to discriminate benign from malicious applications, 3) comparing feature vector models based on system calls, and 4) evaluating multiple machine learning algorithms on the models. The Drebin data set contains 5560 malware applications; after filtering, our malicious data set contains 4289 of those applications.

We obtained the benign data set in February 2014 by utilizing a crawler tool. The tool searched Google Play for free-of-charge applications in all usage categories (e.g., communication, education, music and audio, and business), and randomly collected applications with at least 50,000 downloads. To get a good representation of the Google Play applications while keeping the ratio of malware/goodware at the acceptable level for future classification (see Section 2.3.3), we decided to download roughly three times more goodware applications than the number of obtained malware applications. We stopped our crawler at 12789 collected Google Play applications; after filtering, our benign data set contains 8371 of those applications. Note that we make an assumption that applications with more than 50,000 downloads are benign. The extent to which the assumption is reasonable has a direct impact on the classification results presented in this section. The list of all applications in our input set is published in the MALINE repository.

2.5.2 Configurations

We explore the effects of several parameters in our experiments, where one combination of parameters represents a configuration. The first parameter is the number of events we inject with Monkey into the emulator during an application execution. The number of events is directly related to the length of the execution. We insert 1, 500, 1000, 2000, and 5000 events. It takes 229 seconds on average (with a standard deviation of 106 seconds) for an application execution with 500 events and 823 ($\pm$816) seconds with 5000 events.[6] That includes the time needed to make a copy of a clean virtual machine, boot it, install the application, run it, and download log files from the virtual machine to the host machine.

The second parameter is a flag indicating if a benign background activity should be

[6]The standard deviations are relatively large compared to the averages because some applications crash in the middle of their execution. We take recorded system call traces up to that point as their final execution traces.

present while executing the applications in the emulator. The activity consists of inserting SMS text messages and location updates into the emulator as part of automatic testing. We experiment with the activity only in the 500-Monkey-event experiments, while for all the other experiments, we include no background activity.

It is important to ensure that consistent sequences of events are generated across the executions of all applications. As Monkey generates pseudo-random events, we use the same pseudo-random seed value in all experiments.

We made all data we obtained for different configurations publicly available [DAUR16b].

2.5.3 Results

In this section, we give results of applying random automatic testing on Android applications with the goal of learning a binary classifier that discriminates benign from malicious applications. More precisely, we look at how the type and size of inputs generated by the Monkey testing tool affects the classifier as seen through four quality measures. Finally, we discuss trade offs both in testing and machine learning that have several different impacts on the quality of the classifier.

2.5.3.1 Number of System Calls

As a result of automatic testing by randomly generating and sending events to an application and the Android operating system to drive the execution, numerous system calls are invoked. The total number of system calls an application makes during its execution directly impacts its feature vector, and potentially the amount of information it carries. Hence, we identified the number of injected events, which directly influences the number of system calls made, as an important metric to track. Figure 2.3 shows the number of system calls observed per application in the dynamic analysis phase of an experiment. It can be seen from the figure that the number of system calls observed per application in the dynamic analysis phase of an experiment varies greatly. For example, in an experiment with 500 Monkey events, it ranges from 0 (for applications that failed to install and are filtered out) to over a million. Applications with no system calls are applications that failed to be installed and they were not considered in later analyses. Most of the applications in this experiment had less than 100,000 system calls in total.

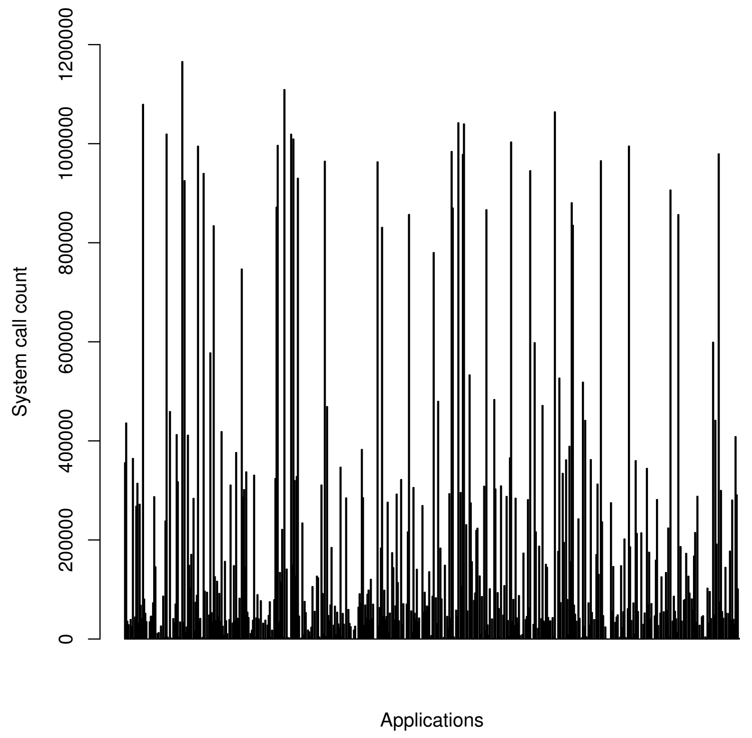

Figure 2.3. Number of system calls per application. The number is for one emulator instance for an experiment with 500 Monkey events and the background activity.

2.5.3.2 Feature Matrices

After the dynamic analysis and feature extraction phases (see Section 2.3) on our filtered input set, MALINE generated 12 different feature matrices. The matrices are based on varying experiment configurations including: five event counts (1, 500, 1000, 2000, 5000), two system call representations (frequency- and dependency-graph-based), and the inclusion of an optional benign activity (SMS messages and location updates) for experiments with 500 events. We refer to these matrices with X_{rep}^{size}, where $rep \in \{freq, graph\}$ is the used representation of system calls and *size* is the number of generated events. In addition, we denote an experiment with the benign background activity using an asterisk.

The obtained feature matrices generated according to the system call dependency representation exhibited high sparsity. This is not surprising since the number of possible system call pairs is 129600. Hence, all columns without a nonzero element were removed from our matrices. Table 2.2 gives the dimensions of the obtained matrices and their level

Table 2.2. Nonzero elements in reduced and full feature matrices. Zero-columns are removed in the reduced matrices.

	Full matrix	**Reduced matrix**			**Full matrix**	**Reduced matrix**	
Type	non-zero (%)	columns	non-zero (%)	Type	non-zero (%)	columns	non-zero (%)
X^1_{freq}	12.48	118	38.09	X^1_{graph}	1.49	11112	17.42
$X^{500}_{freq}*$	17.30	137	45.48	$X^{500}_{graph}*$	3.01	15101	25.83
X^{500}_{freq}	17.27	138	45.07	X^{500}_{graph}	2.99	15170	25.61
X^{1000}_{freq}	17.65	136	46.72	X^{1000}_{graph}	3.12	15137	26.79
X^{2000}_{freq}	17.93	138	46.79	X^{2000}_{graph}	3.22	15299	27.34
X^{5000}_{freq}	18.15	136	48.04	X^{5000}_{graph}	3.29	15262	27.97

of sparsity. Both the frequency and dependency feature vector representations resulted in different nonzero elements in the feature matrices. However, those differences could have only a small or no impact on the quality of classification, i.e., it might be enough only to observe if something happened, which could be encoded as zero/one values. Therefore, we have created additional feature matrices by replacing all nonzero elements with ones to measure the effect of feature matrix structure on the classification.

2.5.3.3 Cross-validated Comparison of Classifiers

Reduced feature matrices (just feature matrices from now on) and goodware/malware labels are inputs to the classification algorithms we used: support vector machines (SVMs), random forest (RF), LASSO, and ridge regression. As described in Section 2.5.1, the Google Play applications were marked as benign and the applications from the Drebin data set as malicious. To avoid possible overfitting, we employed double five-fold cross-validation on the set of applications to tune parameters and test models. To enable comparison between different classifiers for different feature matrices, the same folds were used in the model building among different classification models. Prior to building the final model on the whole training set, all classifiers were first tuned by appropriate model selection techniques to derive the best parameters. The SVMs algorithm in particular required an expensive tuning phase: for each data set, we had to run five-fold cross-validation to find the best C and γ parameters. Hence, we had to run the training and testing phases with different values of C (ranging from 2^{-5} to 2^{15}) and γ (ranging from 2^{-15} to 2^{3}) for the five different splits of training and testing sets. In the end, the best kernel to use with SVMs is the Radial Basis Function (RBF) kernel.

The built classifiers were then validated on the appropriate test sets. If a positive example (i.e., malware in our case) is classified into the positive (respectively, negative) group, we obtained a true positive (respectively, false negative). Analogously, we define true negative and false positive. The threshold for probabilistic classifiers was set at the usual level of 0.5. Since changes to this threshold can have an effect on the sensitivity and the specificity of classifiers, a usual representation of the effect of these changes is given by ROC curves (see Figure 2.4). Here we give ROC curves only for the random forest models (as the best classifiers judging from the cross-validated comparison) with the largest number of events (5000).

Figure 2.5 shows measures of the quality of prediction (see Table 2.1) averaged between cross-validation folds for different classifiers. As it can be seen from the figure, one-event quality measures are consistently the worst in each category, often with a large margin. In other words, the size of input to automatic random testing impacts the quality of resulting classifiers. This indicates the importance of leveraging the information gathered while driving an application using random events. Moreover, the random forest algorithm consistently outperforms all other algorithms across the four quality measures. In the

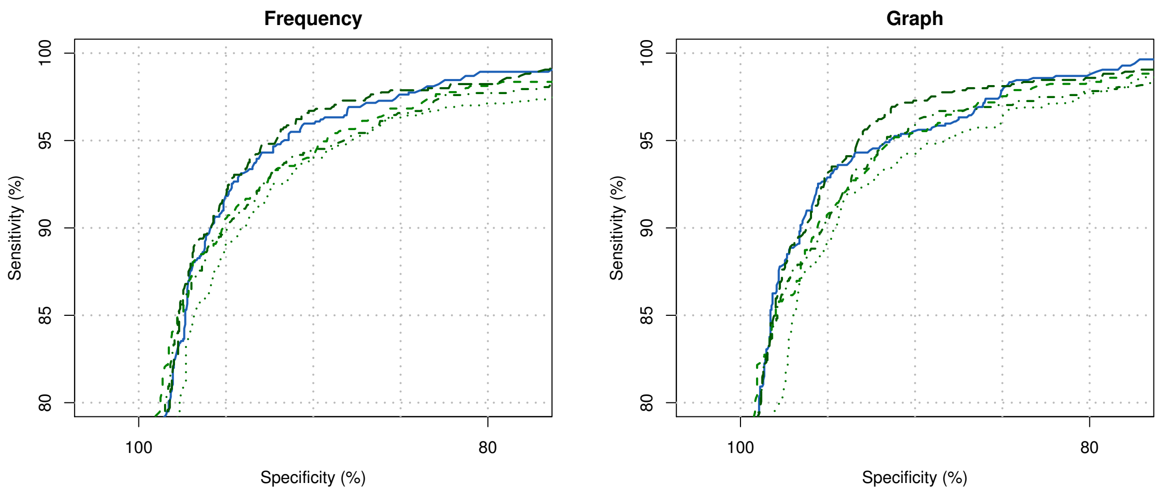

Figure 2.4. ROC curves for five-fold cross-validation with RF model. The curves are for the X_{freq}^{5000} and X_{graph}^{5000} feature matrices.

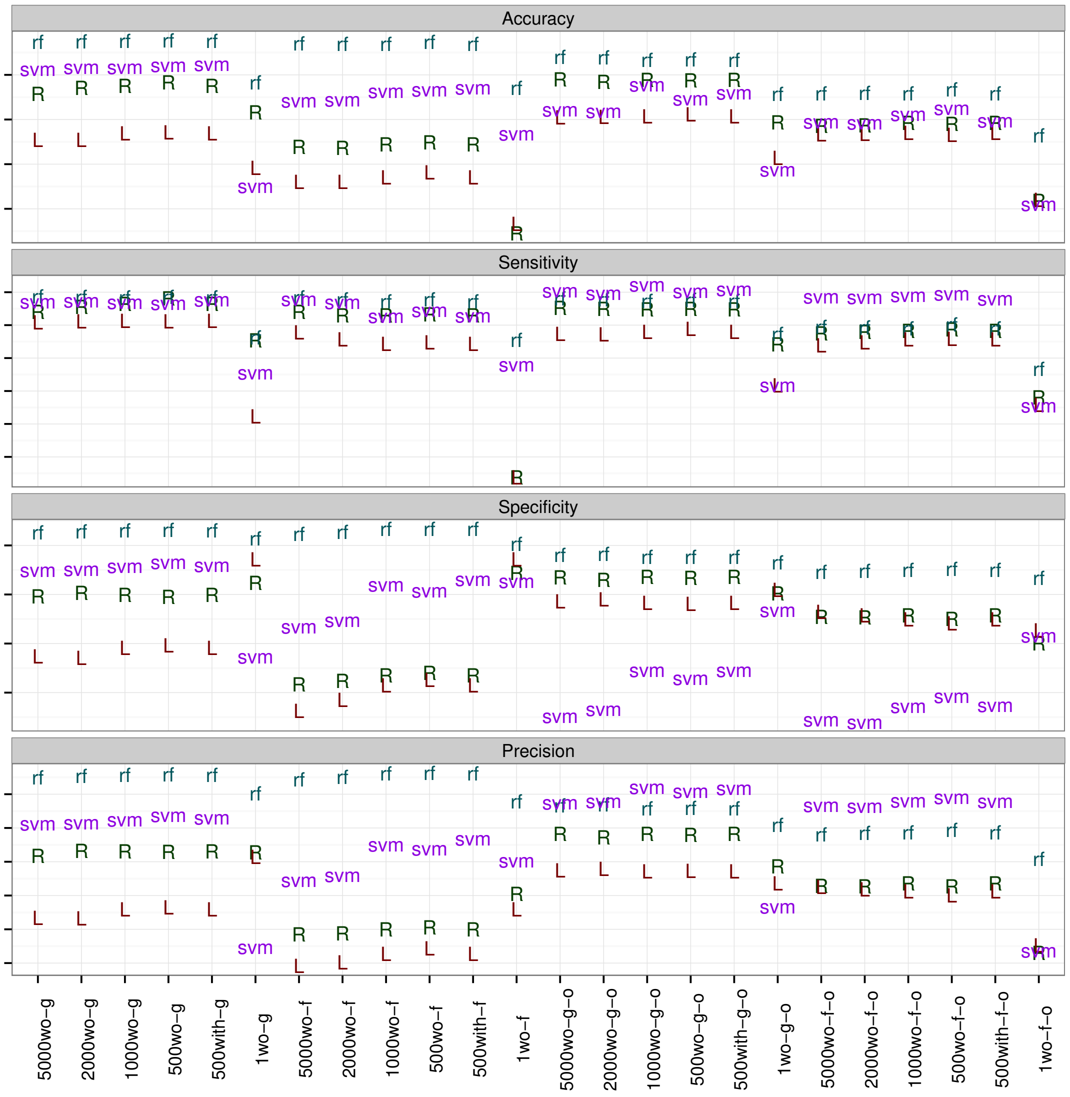

Figure 2.5. Quality comparison of classifiers. All values are averaged on five cross-validation folds. Values on the vertical axis represent percentages. Labels on the horizontal axis are written in the short form where *wo* stands for *without background*, *with* stands for *with background*, *f* stands for *freq*, *g* stands for *graph*, *o* at the end denotes that 0-1 matrices were used, and the numbers at the beginning represent numbers of generated events. 1-event experiments have a 2.3% smaller set of applications.

cases where feature matrices have weights instead of zeros and ones, it shows only small variations across all the input parameters. Other classification algorithms perform better on the dependency than on the frequency representation. Of the other algorithms, SVM is most affected by the presence of the background activity, giving worse sensitivity with the presence, but on the other hand, giving better specificity.

When the weights in the feature matrices are replaced with zeros and ones, thereby focusing on the structure of the features and not their values, all the algorithms consistently perform better on the dependency than on the frequency feature vector representation. However, a comparison within an algorithm based on the weights or zeros and ones in the feature matrices is not straightforward. Random forest clearly performs worse when zeros and ones are used in the feature matrices. LASSO and ridge typically perform better in all the quality measures apart from sensitivity for the zeros and ones compared to the weights.

If a domain expert in Android malware detection is considering to apply MALINE in practice, there are several practical lessons to be learned from Figure 2.5. The expert can choose to use only the random forest algorithm as it consistently provides the best outcomes across all the quality measures. To reduce the time needed to dynamically analyze an application, it suffices to provide 500 Monkey events as an application execution driver. Furthermore, the presence of the benign background activity does not make much of a difference. On the other hand, to provide few execution-driving events to an application does not suffice. Finally, if the time needed to learn a classifier is crucial and more important than sensitivity, the expert can choose the frequency feature vector representation since it yields almost as good results as the dependency representation, but with far smaller feature vectors.

Figure 2.4 shows that there is not much variability between five different folds from the cross-validation of the best-performing algorithm, namely random forest. This indicates a high stability of the random forest model on the input data set regardless of the choice of training and test sets. It is up to the domain expert to make the trade-off choice in tuning a classifier towards either high sensitivity or specificity. The choice is directly related to the cost of having false positives, the benefits of having more true positives, etc. For example, the domain expert may choose the dependency graph feature vector representation and fix the desired specificity level to 95%; from the graph ROC curve in Figure 2.4, it follows that the sensitivity level would be around 93%.

2.5.3.4 Exploring the Effect of Matrix Sparsity

Sparsity of feature matrices can sometimes lead to overfitting. Although we significantly reduce the sparsity with the removal of columns with all zeros, this just removes non-informative features and sparsity is still relatively high (25% for graph representations). To be sure that the effect seen in the cross-validation comparison is real, we performed additional exploration by adopting the idea of permutation tests [OG10].

Due to prohibitively high computational costs, we used only one classification model to explore the effect of sparsity. We chose the random forest classifier, since it gave the best results on the cross-validation comparison and the 5000-event matrices. Prior to building a classifier, we permuted application labels. As before, in this exploration of matrix sparsity, we applied five-fold cross-validation on permuted labels, thus obtaining quality of prediction on the permuted sample. This procedure was repeated 1000 times. Average accuracies of the obtained classifiers were compared to the accuracy of the RF model from Figure 2.5 and they were all significantly lower: the best was at 83% for the system call dependency representation. Although 1000 simulations is not much in permutation models, this exploration reduced the probability of accidentally obtaining high-quality results just because of sparsity.

2.5.3.5 Exploring the Effect of Unbalanced Design

Since the number of malware applications in our input set is half the number of goodware, we have an unbalanced design. Hence, we employed down/up-sampling through bootstrapping to explore if we could get better results using balanced designs (i.e., where the number of malware and goodware applications is the same). Here, we used only the RF classifier to keep computational costs feasible.

Up- and down-sampling exhibited the same effect on the quality of prediction for all feature matrices: it increased sensitivity at the cost of decreased specificity. This comes as no surprise since we have equated the number of malware and goodware applications, thereby giving larger weights to malware applications in the model built compared to the situation before. However, the overall accuracy for models with down-sampling was lower than for the unbalanced model, while for models with up-sampling, it was higher (up to 96.5% for accuracy with a 98% sensitivity and 95% specificity). To explore the stability of results under down- and up-sampling, these methods were repeated 10 times. The standard deviation of accuracies between repeats (on the percentage scale) was 0.302. For a comparison of random forest classifiers with up- and down-sampling, consult Figure 2.6. The figure provides a comparison of random forest classifiers with up- and down-sampling, while Figure 2.7 shows ROC curves for a random forest classifier with up-sampling.

2.5.3.6 Exploring the Effect of Inputs in Testing

As explained in Section 2.3.1, in the dynamic analysis phase of the approach, we use random automatic testing to drive application execution. In general, the type and size of inputs in automatic testing affects the outcome in testing, or in our case, the observed behavior of applications under test. The generated and used inputs in testing have a direct influence on the system calls that happen between an application and the Android operating system, which in turn reflects on feature vectors for machine learning, and finally on the overall quality of binary classifiers for malware detection. The type and size of inputs that we used in evaluating our malware detectors is explained in Section 2.5.2.

Based on the results for four quality measures of machine learning binary classifiers as given in Section 2.5.3.3, we can observe the following with regard to the effect of the type and size of inputs for random automatic testing on the overall quality of malware detection rates. For background activity, which we included only in the case of configurations with 500 Monkey events in order to keep the computation time tractable, there was almost no influence on the quality of the classifiers except for those built with the SVMs algorithm. This background activity, which comprises text messages and location updates, is therefore much less important than internal events generated by Monkey.

When it comes to the size of inputs, i.e., the number of pseudo-random events that Monkey generated and fed to an application under test, we show that, for example, one event only is not sufficient to build high-quality classifiers. On the other hand, the overall quality of built classifiers varies little as we vary the number of Monkey events between 500 and 5000 (see Figure 2.5 and Figure 2.6). This suggests that there is a threshold in terms of the number of Monkey-generated events for learning the behavior of applications. Finally, these insights can provide guidance to malware detection experts in configuring their

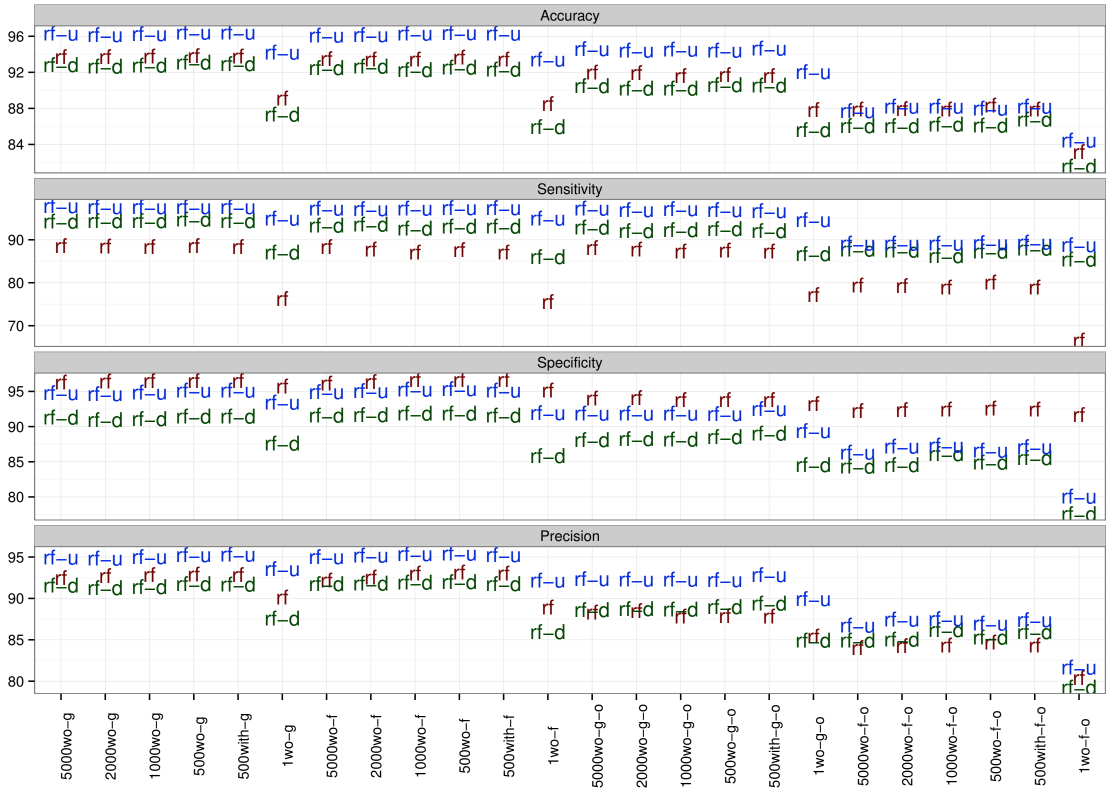

Figure 2.6. Quality comparison of classifiers with up- and down-sampling. All values are averaged on 5 cross-validation folds. Labels on the x axis are written in the short form where *wo* stands for *without background*, *with* stands for *with background*, *f* stands for *freq*, *g* stands for *graph*, *o* at the end denotes that 0-1 matrices were used, and the numbers at the beginning represent number of events used. In the names of classifiers, -*u* denotes up-sampling while -*d* denotes down-sampling.

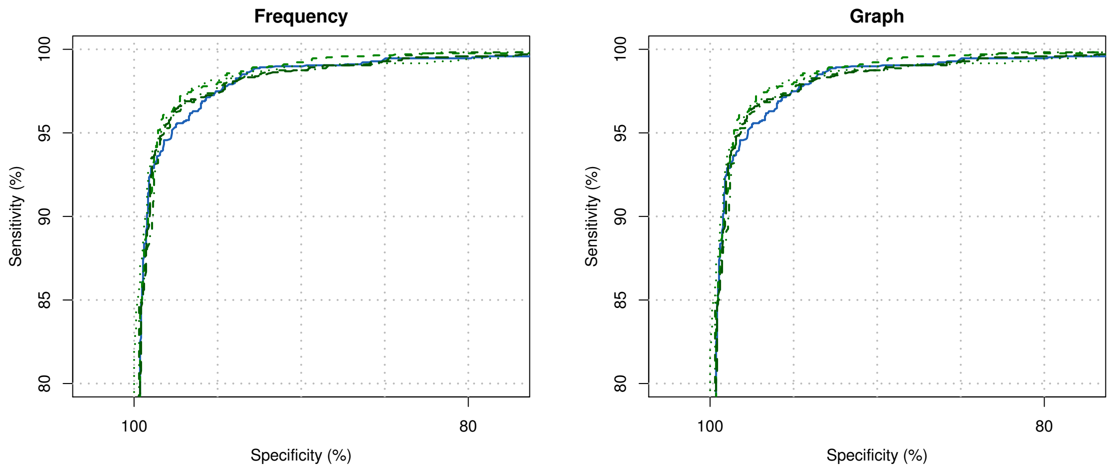

Figure 2.7. ROC curves for five-fold cross-validation with RF model and up-sampling. The curves are for the X_{freq}^{5000} and X_{graph}^{5000} feature matrices with up-sampling.

malware detection systems.

2.5.4 Used Computational Resources

Here we present the scale of used computational resources needed to carry out as thorough and extensive experiments as demonstrated in this chapter. It can be seen that a lot of resources had to be allocated toward such experiments to be able to draw strong conclusions about our approach to malware detection. We employed up to nine 32-core (with hyper-threading up to 64 threads) 128 GB RAM Emulab testbed machines in parallel to reduce the wall-clock time needed to perform the experiments. Not all tasks were executed under the same load, thereby utilizing the available resources to different extents at different stages of the evaluation. We also report memory and disk usage.

The dynamic analysis and feature extraction phases together took around 12 days, typically using over 100 GB of RAM. Executing them on a single-core machine with the same CPU clock speed would have taken about a year. All machine learning techniques, including their training, testing, and evaluation, took around six days; without the parallelization, it would have been more than five years. Simulations for permuted labels took six days to finish and they would have taken almost three years if executed on the single-core machine. To sum up, the total single-core CPU time for all the experiments is approximately nine years, while by heavily parallelizing our experiments, we managed to finish them in less than a month.

The classifiers, including cross-validation, took about five days, which would have been about 50 days. The longest part were the simulations based on permutations.

The first intensive part was running Android emulators and performing the dynamic analysis of applications executing in the emulators. Each virtual machine in an emulator instance would require more than 3 GB of memory to run. Because we had 128 GB RAM computers at hand, we determined 30 instances of Android emulators can execute in parallel on a single testbed computer. Typical usage would be over 100 GB of memory at a given moment. If parallelization had not been employed, it would have taken almost one year of sequential execution on a single computer to analyze applications according to 10 execution combinations of input parameters.

Building a random forest classifier (as described in Section 2.5.3.3) for representation with the system call dependency graph takes around half an hour on 48 cores while for frequency representation, it takes around two minutes. In total, that makes around $48 \times (2 \times 5 + 30 \times 5) = 7680$ minutes or five days and eight hours of CPU time (if one core is used). Using down-sampling takes about 20% less time than for the full model while up-sampling takes around 50% more time. Therefore, if classifiers were to be built on only one core, it would take around 10 days for the cross-validated build of random forest models.

Simulations based on permutations do the same calculations (in permuted labels) as the cross-validated random forest so it takes around the same time to finish. Since we did 1000 simulations, using the average times from the previous paragraph, the duration of the calculations were around $1000 \times 48 \times (30 + 2) = 1,536,000$ minutes or around 2.9 years of CPU time (if one core is used).

Ridge regression took around one hour per five-fold cross-validation on 48 cores while LASSO regression took around 20 minutes. Building random forest, LASSO and ridge classifiers took around five hours, and it would have taken 29 days if executed sequentially on the single-core computer.

The heaviest part of running the SVMs classification against the five different data sets was the parameters selection. Given the resources availability, for each data set, we could run 64 instances of training and find the best parameters in a reasonable amount of time. It took 20 hours to analyze each one of the five data sets containing data about the system calls frequency and 40 hours to analyze the same number of data sets with the dependency calls graph data. Since we ran the experiments in five different machines, the total time was about 60 hours. If we did not have the opportunity to parallelize the experiments using different machines and several cores, running all the experiments sequentially, it would have taken a total of about 600 hours, which would be 25 days of uninterrupted computation.

The total disk usage is also significant. The Android SDK sources, analyzed applications, and generated data during the analyses resulted in 5 TB of disk usage in total.

2.6 Related Work

There is a large body of research on malware detection in contexts other than Android (e.g., [PBCK13, FJC+10, RTWH11, ZJS+11, LBK+10, CAM+08, PMRB09, KCK+09, SDTC+16, JAS14]). While our work was originally inspired by some of these approaches, we primarily focus in this section on more closely related work on Android malware detection. Ever since Android has become popular, there has been an increasing body of research on detecting malicious Android applications, and we split that research into static and dynamic analysis techniques.

2.6.1 Static Techniques

Static techniques are typically based on source code or binary analyses that search for malicious patterns (e.g., [FADA14, WROR14]). For example, static approaches include analyzing permission requests for application installation [ADY13, GTGZ14, FHE+12], control flow [LKM+13, LSM14], signature-based detection [GZZ+12, FADA14], and static taint-analysis [ARF+14].

Stowaway [FCH+11] is a tool that detects over-privilege requests during the application install time. Enck et al. [EOMC11] study popular applications by decompiling them back into their source code and then searching for unsafe coding security issues. Yang et al. [YXA+15] propose AppContext, a static program analysis approach to classify benign and malicious applications. AppContext classifies applications using machine learning based on the contexts that trigger security-sensitive behaviors. It builds a call graph from an application binary and after different transformations, it extracts the context factors via information flow analysis. It is then able to obtain the features for the machine learning algorithms from the extracted context. In the paper, 202 malicious and 633 benign applications from the Google Play store are analyzed. AppContext correctly identifies 192 malicious applications with an 87.7% accuracy. Gascon et al. [GYAR13] also use call graphs to detect malware. Once they extract function call graphs from Android applications, they apply a linear-time graph kernel in order to map call graphs to features. These features are given as input to SVMs to distinguish between benign and malicious applications. They conducted experiments on 135,792 benign and 12,158 malware applications, detecting 89% of the malware with a 1% false positive rate.

2.6.2 Dynamic Techniques

Dynamic analysis techniques consist of running applications in a sandbox environment or on real devices in order to gather information about the application behavior. Dynamic taint analysis [EGC$^+$10, YY12] and behavior-based detection [DMSS12, BZNT11] are examples of dynamic approaches. Our approach analyzes Android applications dynamically and captures their behavior based on the execution pattern of system calls. Some existing works follow similar approaches.

Dini et al. [DMSS12] propose a framework MADAM for Android malware detection, which monitors applications at the kernel and user level. MADAM detects system calls at the kernel level and user activity/idleness at the user level to capture the application behavior. Their extremely preliminary and limited results, considering only 50 goodware and two malware applications, show a 100% detection accuracy. Crowdroid [BZNT11] is another behavior-based Android malware detector that uses system calls and machine learning. As opposed to our approach, Crowdroid collects information about system calls through a community of users. A lightweight application, installed in the users' devices, monitors the system calls (frequency) of running applications and sends them to a centralized server, which performs classification. Crowdroid was evaluated on a limited number of goodware applications and only two malware applications, obtaining detection accuracies of 100% for one and 85% for the other.

Rieck et al. [RTWH11] propose another framework for automatic Windows malware detection that leverages applications behavior and machine learning. The proposed framework analyzes each application and maps its monitored behavior (through system call tracing) to feature vectors that can be used in both clustering and classification techniques. The framework groups system calls based on their functionality, thereby forming a hierarchical structure for their feature vector representation. By alternating clustering and classification, the framework can incrementally analyze thousands of applications on a daily basis and identify novel and known classes of malware. Known malware classes are identified first. Then, the obtained information about known malware classes combined with unidentified application behavior reports is clustered for discovery of new malware classes. Their experimental results show an F-measure of 96%, while our computed F-measure is 91%.

2.7 Threats to Validity

There are multiple ways in which validity of this work could be compromised. We address them here.

2.7.1 Application Crashes

We observed applications crashing while performing our experiments, which could bias our empirical results. This might impact conclusions we draw about what kind of behavior is learned with machine learning. In particular, it could be that one behavior group crashes more often; hence, we would be learning to discriminate a more-crashing from a less-crashing group of applications, and not goodware from malware. A crash in general happens due to an application ending up in a program state not foreseen by its developer, resulting in a non-regular application termination. We used the Monkey tool in the experiments to drive applications in their executions. Given that Monkey generates sequences of pseudo-random input events, it is to be expected that it can drive an application into a state that does not handle certain kinds of events, causing a crash. Depending on an experiment, we observed from 29% to 49% applications crash, which could bias our empirical results. However, the crash rate of goodware and malware applications is roughly the same. Therefore, application crashes do not bring in a classification bias.

2.7.2 Age of Applications

Our goodware data set comprises applications downloaded in 2014, while our malware applications are from 2010 – 2012. Because the Android operating system's API evolved from 2010 to 2014, it could mean our approach learns differences between APIs, and not differences between benign and malicious behaviors. Unfortunately, we could not obtain older versions of applications from Google Play as it hosts only the most recent versions. In addition, to the best of our knowledge, a more recent malware data set does not exist. Hence, we manually downloaded 2010 – 2012 releases of 92 applications from F-Droid [fdr15], an Android application repository offering multiple releases of free software applications; we assumed the applications to be benign. We classified them using MALINE, and we obtained specificity of around 88%. Compared to the specificities from Figure 2.5, which were typically around 96%, this might indicate that MALINE performs API difference learning to some extent. However, a comparison with a much bigger set of the same applications across different releases would need to be performed to draw strong conclusions. This suggests that the difference in age of applications used in our experiments does not create a considerable bias.

2.7.3 Hidden Malicious Behavior

Malicious behavior may occasionally be hidden and triggered only under very specific circumstances. As our approach is based on random testing, we might miss such hard-to-reach behaviors, which could affect our ability to detect such application as malicious. Such malware is not common though, and ultimately, we consistently obtain sensitivity of 87% and more using MALINE.

2.7.4 Detecting Emulation

As noted in previous work [JZAH14, PMRB09, CAM$^+$08], malware could potentially detect it is running in an emulator and alter its behavior accordingly. MALINE does not address this issue directly. However, an application trying to detect that it is being executed in an emulator triggers numerous system calls, which likely leaves a specific signature that can be detected by MALINE. We consistently obtain sensitivity of 87% and more using MALINE. If we are to assume that all the remaining malware went undetected only due to its capability of detecting the emulator and consequently changing its behavior without leaving the system call signature, it is at most 13% of the malware in our experiments that successfully disguise as goodware. Finally, Chen et al. [CAM$^+$08] show that less than 4% of the malware in their experiments changes its behavior in a virtualized environment.

2.7.5 System Architecture and Native Code

While the majority of Android-powered devices are ARM-based, MALINE uses an x86-based Android emulator for performance reasons. Few Android applications — less than 5% according to Zhou et al. [ZWZJ12] — contain native libraries typically compiled for multiple platforms, including x86, and hence they can be executed with MALINE. Nonetheless, the ARM and x86 system architectures have different system calls: with the x86-based and ARM-based emulator, we observed applications utilizing 360 and 209 different system calls, respectively. Our initial implementation of MALINE was ARM-based, and switching to an x86-based implementation yielded roughly the same classification results in preliminary experiments, while it greatly improved performance.

2.7.6 Randomness in maline

In MALINE we used only one seed value for Monkey's pseudo-random number generator; it is possible the outcome of our experiments would have been different if another seed value was used. However, as the seed value has to be used consistently within an experiment consisting of thousands of applications, it is highly unlikely the difference would be significant.

2.8 Conclusions

In this chapter, we presented a free software reproducible research environment MALINE for dynamic-analysis-based malware detection in Android. Our approach is based on leveraging random automatic testing to generate events that drive an application execution in Android, observing system calls that occur during the execution, encoding the system calls into features for machine learning, and finally building binary classifiers that discriminate benign from malicious applications. We presented an extensive empirical evaluation of our novel system call encoding into a feature vector representation against a well-known frequency representation across several dimensions. The novel encoding shows better quality than the frequency representation. Our evaluation provides numerous insights into the structure of application executions, the impact of different machine learning techniques, and the type and size of inputs to automatic testing in dynamic analyses, serving as a guide for future research. To facilitate further and reproducible research, we made our data freely available.

In this work, we showed how the type and size of inputs in automatic testing affect the quality measures of built binary classifiers. Background activity such as text messages and location updates usually have negligible influence, except for classifiers built with the SVMs algorithm. Therefore, for almost all classifiers, internal events sent directly to applications under test had the defining impact. In terms of the size of inputs, very few inputs are not enough to build good classifiers. However, the difference in the quality measures between a few hundred and a few thousand random events in automatic testing is very small. This shows that efficient malware detection can be achieved with a relatively short amount of time spent in automatic random testing of applications under consideration.

CHAPTER 3

AUTOMATIC TESTING FOR OBJECT-ORIENTED SOFTWARE

This chapter is based on several publications [Dim13, DGH$^+$14, DR13, LDG$^+$16].[1]

3.1 Introduction

Software developers heavily rely on testing for improving the quality of their software. There are good reasons for adopting this practice. First, as opposed to more heavyweight techniques such as static analysis, testing is easy to deploy and understand, and most developers are familiar with software testing processes and tools. Second, testing is scalable (i.e., millions of tests can be executed within hours even on large programs) and precise (i.e., it does not generate false alarms that impede developers' productivity). Third, while testing cannot prove the absence of bugs, there is ample evidence that testing does find important bugs that are fixed by developers. Despite these advantages, testing is not a silver bullet since crafting good tests is a time-consuming and costly process, and even then, achieving high coverage and catching all defects using testing can be challenging. Naturally, there has been a great deal of research on alleviating these problems by developing techniques that aim to improve the automation and effectiveness (in terms of achieved coverage and defects found) of software testing.

Random testing is the most basic and straightforward approach to automating software testing. Typically, it completely automatically generates and executes millions of test cases within hours, and quickly covers many statements/branches of the software under test (SUT). However, a drawback of random testing is that, depending on the characteristics of the SUT, the achieved coverage plateaus due to unlikely execution paths. Figure 3.1 gives our motivating example Java program that illustrates this point. To apply random testing

[1]Portions of the published work are reused and reprinted here with permission.

```
public class Absolute {
  private int x;
  public Absolute(int x) {
    this.x = x;
  }
  public int difference(int y) {
    int out;
    if (x > y) out = x - y;
    else out = y - x;
    assert out > 0;
    return out;
  }
  @Test public void testAbsolute() {
    Absolute abs = new Absolute(10);
    abs.difference(0);
  }
}
```

Figure 3.1. Example JAVA program for computing absolute difference. It consists of class `Absolute` and its method `difference` that computes the absolute difference between field `x` and input parameter `y`. It also checks whether the computed difference is greater than 0. In addition, we include a simple unit test for this class and method.

on the example, we generate the following randomized unit test:

```
public void testAbsolute() {
  Absolute abs = new Absolute(random());
  abs.difference(random());
}
```

Clearly, it is trivial to execute this simple unit test many times, each time with a new pair of random numbers being generated. It is unlikely, however, that executing it would generate inputs that violate the assertion swiftly since that requires for the two inputs to be equal; moreover, the achieved code coverage would plateau. A quick analysis of the code reveals that covering the assertion amounts to solving a simple logical constraint over inputs of the form $X \leq Y \wedge Y - X \leq 0$ (see Section 3.2.1). This observation is the basis for dynamic symbolic execution, which leverages automatic constraint solvers to compute test inputs that cover such hard-to-cover branches. For example, the JDART [LDG$^+$16] dynamic symbolic execution tool when run on method `testAbsolute` generates test cases covering all branches in less than a second, thereby triggering an assertion violation. A paper on JDART [LDG$^+$16] also shows that the tool improves coverage over random testing for a class of numerically intensive SUTs. In general, symbolic testing-based methods excel in automatically generating test inputs over primitive numeric data types, and have hence been successfully applied as either system-level (e.g., SAGE [GLM12], KLEE [CDE08]) or method-level (e.g., JDart [LDG$^+$16], JCute [SA06]) test generators.

Generating unit tests for object-oriented software poses an additional challenge: instead of taking just primitive types as input, methods in object-oriented software require a rich heap structure of class objects to be generated. We can observe this even in the simple unit test given in Figure 3.1. Here, testing of method `difference` requires an object of type `Absolute` to be first created and initialized, and in turn, `difference` is invoked on it. While several approaches have been proposed that automatically generate symbolic heap structures [KPV03], logical encoding of such structures results in more complex constraints that put an additional burden on constraint solvers; hence, these approaches have not yet seen wider adoption on large SUTs. On the other hand, generating heap structures by randomly creating sequences of constructor+method invocations was shown to be effective, in particular when advanced search- and feedback-directed algorithms are employed (e.g., EvoSuite [FA11], Randoop [PLEB07]). It is then natural to attempt to combine the two approaches by using random testing to perform global/macro exploration (by generating heap structures using sequences of constructor+method invocations at the level of classes) and dynamic symbolic execution to perform local/micro exploration (by generating inputs of primitive types using constraint solvers at the level of methods). In this chapter, we describe, implement, and empirically evaluate such a hybrid approach.

Our hybrid approach combines feedback-directed unit testing with dynamic symbolic execution. We leverage feedback-directed unit testing to generate method sequences that create heap structures and drive a SUT into interesting global (i.e., macro) states. We feed the generated sequences to a dynamic symbolic execution engine to compute inputs of primitive types that drive the SUT into interesting local (i.e., micro) states. We implemented this approach in a tool named JDoop, which combines the feedback-directed unit testing tool Randoop [PLEB07] with our dynamic symbolic execution engine JDart [LDG$^+$16]. Given that such a combination has not been thoroughly empirically studied in the past, we also assess the merits of this approach through a large-scale empirical evaluation.

Our main contributions are as follows:

- We developed JDoop, a hybrid tool that combines feedback-directed unit testing with dynamic symbolic execution to be able to experiment with large-scale automatic testing of object-oriented software.

- We implemented a distributed benchmarking infrastructure for running experiments in isolation on a cluster of machines. This allows us to execute large-scale experiments that ensure statistical significance, and also advances the reproducibility of our results.

- We performed an extensive empirical evaluation and comparison between random (our baseline) and hybrid testing approaches in the context of automatic testing of object-oriented software.

- We identified several open research questions during our evaluation, performed additional targeted experiments to obtain answers to these questions, and provided guidelines for future research efforts in this area.

3.2 Preliminaries

In this section, we introduce dynamic symbolic execution, feedback-directed random testing, and explain how in particular we implemented dynamic symbolic execution in a modular testing framework called JDart.

3.2.1 Dynamic Symbolic Execution

Dynamic symbolic execution [GKS05, SMA05, CDE08] is a program analysis technique that executes a program with concrete and symbolic inputs at the same time. It systematically collects constraints over the symbolic program inputs as it is exploring program paths, thereby representing program behaviors as algebraic expressions over symbolic values. The program effects can thus be expressed as a function of such expressions.

Dynamic symbolic execution maintains, in addition to the concrete state defined by the concrete program semantics, the symbolic state, which is a tuple containing symbolic values of program variables, a path condition, and a program counter. A path condition is a conjunction of symbolic expressions over the symbolic inputs that characterizes an execution path through the program. It is generated by accumulating (symbolic) conditions encountered along the execution path, so that concrete data values that satisfy it can be used to drive its concrete execution. Such values are typically computed using automatic constraint solvers. Path conditions are stored as a symbolic execution tree that characterizes all the paths exercised as part of the symbolic analysis.

In dynamic symbolic execution, the symbolic execution tree is built by repeatedly augmenting it with new paths that are obtained from unexplored branches in the tree. This is done by employing an exploration strategy such as depth-first, breadth-first, or random. A constraint solver is used to obtain a valuation for a yet-unexplored branch by feeding it the corresponding path condition. The new valuation drives a new iteration of dynamic symbolic execution that augments the symbolic execution tree with a new path.

Figure 3.1 gives a simple JAVA program that we use to illustrate how dynamic symbolic execution works. Note that the provided unit test does not fail when executed. However, the assertion can in fact be violated when `x` and `y` are equal, and we describe next how dynamic symbolic execution generates such inputs. First, we treat field `x` and parameter `y` as symbolic inputs. Their values, as well as all decisions involving them, are recorded during execution as symbolic constraints. We use X and Y to represent the symbolic inputs. The resulting symbolic execution tree is shown in Figure 3.2.

In the initial state, the path condition PC is $True$. The algorithm then proceeds with the first concrete execution that uses the initial provided concrete inputs `x = 10` and `y = 0`. The if-condition in method `difference` involves symbolic values that are recorded in the path condition. Since `x > y` and variable `out = 10`, the assertion in the unit test is not violated

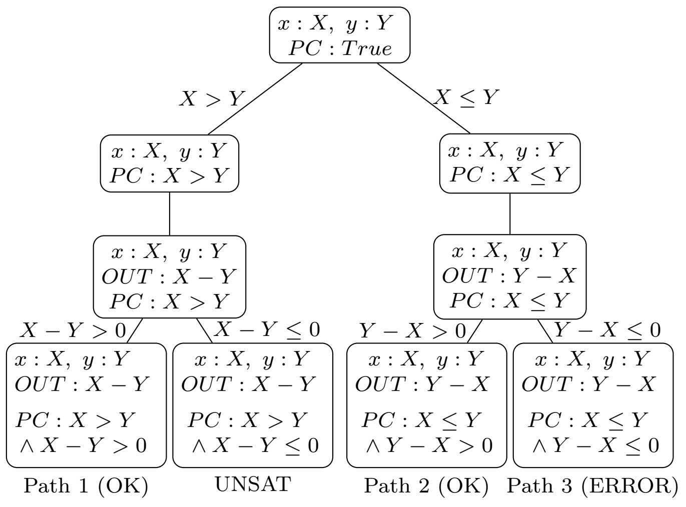

Figure 3.2. Symbolic execution for program in Figure 3.1. The tree characterizes the paths exercised by dynamic symbolic execution of the example program.

and its condition is similarly recorded in the path condition. Then, the execution terminates (Path 1). To drive the next iteration, the algorithm uses a constraint solver to try to compute concrete values for exploring a yet-unexplored branch: the false branch of the condition in the assertion. This path is represented by the path condition $X > Y \wedge X - Y \leq 0$, which is unsatisfiable, and hence this path is not feasible.

The algorithm moves to the next unexplored branch: the false branch of the if-condition with path condition $X \leq Y$. Let us assume that a solver will return the satisfying assignment `x = 5` and `y = 10`. A new iteration of dynamic symbolic execution is driven by these concrete values, resulting in Path 2 being exercised (still no assertion violation). The final unexplored branch is the false branch of the condition in the assertion represented by the path condition $X \leq Y \wedge Y - X \leq 0$. Indeed this constraint is satisfiable whenever $X = Y$, which leads to an assertion violation. Since there are no other unexplored branches left in the symbolic execution tree, the algorithm terminates.

3.2.2 Feedback-directed Random Testing

A simple approach to automatic unit testing of object-oriented software is to completely randomly generate sequences of constructor+method invocations together with the respective concrete input values. However, this typically results in a large overhead since numerous sequences get generated with invalid prefixes that lead to violations of common implicit class or method requirements (e.g., passing a null reference to a method that expects an allocated object). Moreover, such sequences cause trivial, uninteresting exceptions to be thrown early, thereby preventing deep exploration of the SUT state space. Hence, instead of generating unit tests blindly and in a completely random fashion, useful feedback can be gathered from previous test executions to direct the creation of new unit tests. In this way, unit tests that execute long sequences of method calls to completion (i.e., without exceptions being thrown) can be generated. This approach is known as feedback-directed random testing and is implemented in the RANDOOP automatic unit testing tool [PLEB07].

RANDOOP uses information from previous test executions to direct further unit test generation. The tool maintains two sets of constructor+method invocation sequences: those that do not violate a property (i.e., property-preserving) and those that do (i.e., property-violating). The property-violating set is initially empty, while the property-preserving set is initialized with an empty sequence. The default property that is maintained is unit test termination without any errors or exceptions being thrown. Randoop randomly selects a public method (or a constructor) and an existing sequence from the property-preserving set. It then appends the invocation of the selected constructor/method to the end of the sequence, and replaces primitive type arguments with concrete values that are randomly selected from a preset pool of values. Next, the newly generated sequences are compared against all previously generated sequences in the two sets. If it already exists, it is simply dropped and random selection is repeated. Otherwise, Randoop executes the new sequence and checks for property violations. If no properties are violated, the sequence is added to the property-preserving set and otherwise to the property-violating set. Randoop keeps on extending property-preserving sequences until it reaches a provided time limit.

3.2.3 JDart

JDart [LDG+16] is a modular testing framework for Java bytecode. The development of JDart has been driven by two main goals. The primary goal has been to build a symbolic analysis framework that is robust enough to handle large-scale software. More precisely, it has to be able to execute such software without crashing, deal with long execution paths, and deal with complex path constraints. The second objective has been to build a modular and extensible platform that can be used for the implementation and evaluation of novel ideas in dynamic symbolic execution. For example, JDart is designed to allow for easy replacement of all of its components: it supports different and combined constraint solvers, and several exploration strategies and termination criteria.

A run of JDart produces the following outputs: a symbolic execution tree that contains all explored paths along with performance statistics, vectors of concrete input values that execute paths in the tree, and a suite of test cases (based on these vectors). A symbolic execution tree contains leaf nodes for all explored paths (similar to the one shown in Figure 3.2) and additionally leaves for branches off of executed paths that could not be explored because the constraint solver was not able to produce adequate concrete values or because native code is not executed in the fully symbolic mode (JDart's ability to handle native code is described later in Section 3.2.3.5). For these leaves, JDart does not generate input vectors or test cases.

This section presents the modular architecture of JDart, and discusses its main components and extension points. It subsequently describes existing uses of JDart as a component within other research tools.

3.2.3.1 Architecture

JDart executes Java bytecode programs and performs a dynamic symbolic analysis of specific methods in these programs. JDart also implements extensions that build upon the results of a dynamic symbolic analysis:

- the Method Summarizer generates fully abstract method summaries for analyzed methods [HGR13]. In the generated summaries, class members, input parameters, and return values are represented symbolically.
- the Test Suite Generator generates JUnit test suites that exercise all the program paths found by JDart.

During dynamic symbolic analysis, JDart uses two main components to iteratively execute the target method, to record and explore symbolic constraints, and to find new concrete data values for new executions. Figure 3.3 depicts the modular architecture of JDart. The basis (at the bottom) is the Executor that executes the analyzed program and records symbolic constraints on data values. The Explorer organizes recorded path constraints into a constraints tree, and decides which paths to explore next, and when to stop exploration. The Explorer uses the JConstraints library to integrate different constraint solvers that can be used in finding concrete data values for symbolic paths constraints.

3.2.3.2 Executor

The Executor runs a target program and executes an analyzed method with different concrete data values for method parameters and class members. It also records symbolic constraints for program paths. Currently, JDart uses the software model checker Java PathFinder (JPF) for the execution of Java bytecode programs. JDart uses two extension points of JPF.

JPF uses “choice generators” to mark points in an execution to which JPF backtracks during state-space exploration. JDart implements a choice generator that sets parameter values of methods that are analyzed symbolically.

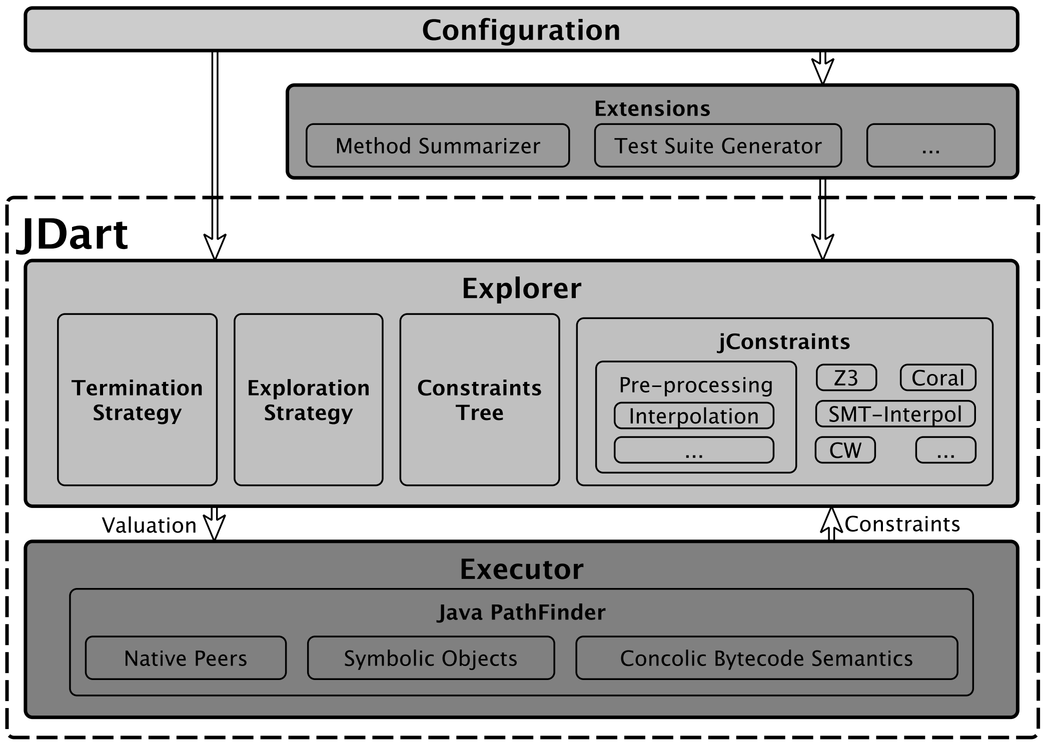

Figure 3.3. Architecture of JDart.

JPF extensions can provide custom bytecode implementations. JDart adds a concolic semantics to Java bytecodes that performs concrete and symbolic operations simultaneously, while also recording path constraints. Using JPF as an execution platform has several benefits. For example, it is easy to integrate other JPF extensions in JDart (e.g., for dealing with native code or for recording test coverage). Moreover, JPF provides easy access to all objects on the heap and stack, as well as to many other elements and facilities of the JVM such as stack frames and class loading.

On the other hand, using a full-blown custom JVM for execution has an impact on performance. This is one of the reasons why we keep the integration with JPF as loose as possible. JDart has been built with the possibility of changing the underlying execution environment from JPF to a more lightweight instrumentation, as is the case with other similar frameworks such as PEX [TH08] and JCute [SA06].

3.2.3.3 Explorer

The Explorer organizes recorded constraints into a constraints tree, decides which parts of the program to explore, when to stop, and how to solve constraints for new concrete input values.

To hit interesting paths quickly when analyzing large systems, JDart needs to be able to limit exploration to certain paths. JDart provides configuration options for specifying multiple predetermined vectors of input values from which the exploration is started. It also allows the user to specify assumptions on input parameters as symbolic constraints. JDart will then only explore a method within the limits of those assumptions. Finally, JDart can be configured to skip exploration of certain parts of a program (e.g., after entering a specific method), i.e., it supports suspending and resuming exploration based on method level descriptions. It also allows skipping exploration after a certain depth.

For large-scale systems, it is often not possible to run an analysis to completion. Sometimes, one may even be interested in recording the path constraint of a single program path (cf., e.g., SAGE [GLM12]). JDart provides an interface for implementing customized termination strategies. So far, it provides strategies for terminating after a fixed number of paths and for terminating after a fixed amount of time.

In sizable software systems, path constraints can be long and complex and may contain trigonometric or elementary functions, which may challenge any advanced constraint solver. JDart provides several techniques and extension points for optimizing constraints, e.g., by simplifying path constraints, adding auxiliary definitions and/or interpolation that help solving complex constraints, and using specialized solvers. These capabilities are based on the constraint processing features of JConstraints. For example, trigonometric constraints can be approximated by interpolation before being submitted to a solver (e.g., Z3) or they can be delegated directly to a solver that supports them (e.g., Coral).

Floating-point constraints can also be processed before submitting them to a solver. For the Z3 integration, floating-point constraints are approximated using reals. Despite this not being sound (due to the limited-precision effects), it might frequently yield valuable solutions even when they are incorrect. In general, JDart always analyzes the solutions and tests whether they can be used to exercise previously unexplored paths.

Finally, it is important to guarantee that progress is made when only approximating the Java semantics in solvers. Sometimes, a solution suggested by a solver may not be valid for a Java bytecode program. JDart tests all valuations produced by a decision procedure on the constraints tree by evaluating path constraints with the Java semantics before re-executing the program with a new valuation. (This is a feature provided by JConstraints, as explained later in this section.)

3.2.3.4 JConstraints

JConstraints is a constraint solver abstraction layer for Java. It provides an object representation for logic expressions, unified access to different SMT and interpolation solvers, and useful tools and algorithms for working with constraints. While JConstraints was developed for JDart, it is maintained as a stand-alone library that can be used independently. The idea has been explored by others, e.g., PySMT [GM15], which was developed for the Python programming language.

The architecture of JConstraints is shown in Figure 3.4. It consists of the basic library providing the object representation of logic and arithmetic expressions, the API definitions for solvers (for SMT solving and interpolation, or for incremental solving), and some basic utilities for working with expression objects (basic simplification, term replacement, and term evaluation). Plugins for connecting to different constraint solvers can be added easily by implementing a solver interface and taking care of translating between a solver-specific API and the object representation of JConstraints.

Currently, plugins exist for connecting to the SMT solver Z3 [dMB08], the interpolation solver SMTInterpol [CHN12], the meta-heuristic-based constraint solver Coral [SBdP11], and a solver that implements the concolic walk algorithm [DA14]. JConstraints uses the native interfaces for these solvers as they are much faster than the file-based integration. It can also parse and export constraints in its own format and supports a subset of the SMT-LIB format [smt17], which enables connection to many constraint solvers that support

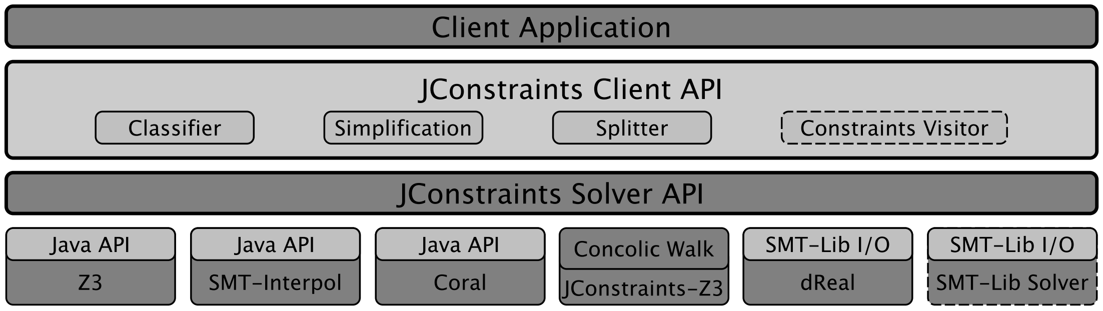

Figure 3.4. Architecture of JConstraints.

this format. For example, through the SMT-LIB format, we were able to experiment with using the dReal solver [GKC13] for nonlinear constraints in JDart.

JConstraints supports both Java and user-defined types for expressions. This enables it to record path constraints directly in terms of the analyzed program types and semantics, as opposed to the types supported by the constraint solver to be used. An advantage of this feature is that it is easy to validate solutions returned by constraint solvers by simply evaluating the path constraint stored by JConstraints with the Java semantics.

3.2.3.5 Handling Native Code

A limitation of JDart's approach to symbolic execution of Java programs is that native code is outside the scope of the analysis. Based on the Nhandler extension [SB14] to JPF, JDart offers two strategies for dealing with native code:

- **Concrete Native.** In this mode, JDart executes native code on concrete data values, and no symbolic execution of native parts is performed. Only concrete values are passed to and from native calls, and symbolic values are not updated and cannot taint native return values. The return value is annotated with a new symbolic variable. As a consequence, the concrete side of an execution is faithful to the respective execution on a normal JVM. However, branches in the native code are not recorded in symbolic path conditions, which can lead to JDart not being able to explore branches after a native call. Another downside of this mode is that the implementation in JPF is relatively slow.

- **No Native.** In this mode, JDart does not execute native code at all. Instead, it returns a default concrete value every time a native method is called and a return value is expected. The concrete value is annotated with the corresponding symbolic variable, using the method signature of the native method as the name of that variable. Concrete execution, in this case, is not faithful to the respective execution on a normal JVM as the introduced default values in most cases are not equal to the values that would be returned by the actual method invocations (and side effects are ignored as well). Recorded symbolic branches cannot be explored even if solutions are found by a constraint solver as there currently is no mechanism that allows feeding these values into the execution (instead of the default return values of native methods).

Since the 'No Native' mode is more performant and since currently there is no way of solving most of the recorded constraints in 'Concrete Native' mode (cf. results in Section 3.4), JDoop runs JDart in 'No Native' mode for native code. We use the 'Concrete Native' mode in our evaluation for analyzing the potential limiting impact of not executing native code faithfully and not being able to find and inject values that target branches in native code.

3.3 Hybrid Approach

In this section, we describe our hybrid approach that combines dynamic symbolic execution and feedback-directed random testing into an algorithm for automatic testing of object-oriented software. We implemented this algorithm as the JDoop tool that is freely available online under the GNU General Public License version 3 or later at `https://github.com/psycopaths/jdoop`. Figure 3.5 shows the flow of the algorithm, which is iterative and each iteration consists of several stages that we describe next. Furthermore, each stage is time-limited by a time parameter, where the second and third stage described below have a common time limit.

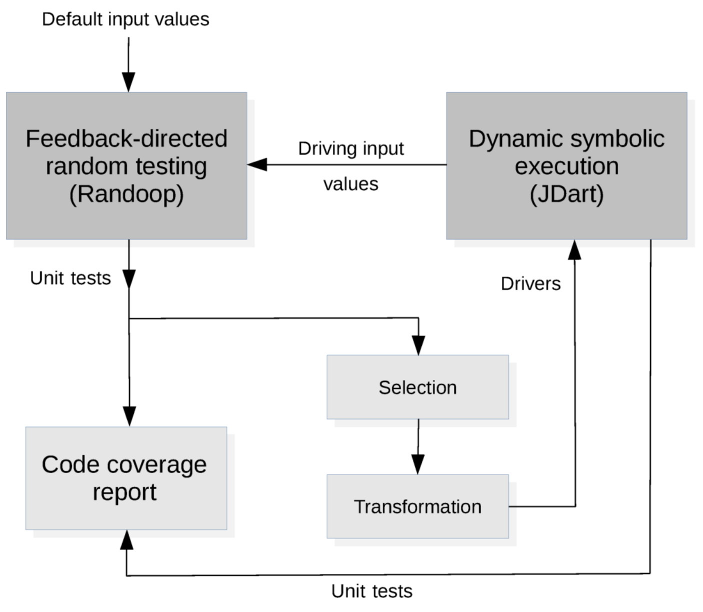

Figure 3.5. Iterative algorithm of JDoop for unit test generation. The algorithm combines dynamic symbolic execution and feedback-directed random testing.

3.3.1 Generation of Sequences

The first stage of every iteration of our algorithm is feedback-directed random testing using Randoop, which generates constructor+method sequences as described in Section 3.2.2. Randoop takes advantage of a pool of concrete primitive values to be used as constructor/method arguments when generating sequences. In the first iteration, we use the default pool with few values, which for the integer type are -1, 0 1, 10, 100. Hence, an instance of a generated sequence for our running example from Figure 3.1 is the following:

```
void test1() {
  Absolute abs = new Absolute(100);
  abs.difference(1);
}
```

Our algorithm grows the pool for subsequent iterations with concrete inputs generated by dynamic symbolic execution, which we describe later. The sequences generated in this stage serve two purposes. First, we employ them as standalone unit tests that exercise the SUT, which is their original intended purpose. Second, our hybrid algorithm also employs them as driver programs to be used in the subsequent dynamic symbolic execution stage.

3.3.2 Selection and Transformation of Sequences

The previous stage typically generates far too many sequences to be successfully explored with a dynamic symbolic execution engine in a reasonable amount of time. For example, several thousand valid sequences are often generated in just a few seconds. Hence, it is prudent to select a promising subset of the generated sequences to be transformed into inputs for the subsequent dynamic symbolic execution with JDart. The second stage implements the selection and transformation of constructor+method sequences.

Note that dynamic symbolic execution techniques have limitations, which is why we implemented the hybrid approach in the first place. In particular, they can typically treat symbolically only method arguments of primitive types. For example, if a sequence contains method calls with non-primitive types only, JDart will not be able to explore any additional paths. Hence, not every generated sequence is suitable for dynamic symbolic execution with JDart, and as the first step of this stage, we filter out all sequences with no arguments of a primitive type. Next, we have two strategies (i.e., heuristics) for selecting promising sequences. The first strategy randomly selects a subset of sequences. The second strategy prioritizes candidate sequences with more symbolic variables, which is based on the intuition that having more symbolic variables leads to more paths (and also branches and instructions) being covered. We compare the two strategies in our empirical evaluation. Once promising sequences are selected, they have to be appropriately transformed into driver programs for JDart.

Every candidate sequence is transformed for the final stage where dynamic symbolic execution is performed. This is achieved by turning all constructor/method arguments of primitive types, which are supported by JDart, into symbolic input values. In our implementation, this is a simple source-to-source transformation. For instance, our example sequence results in the following driver program:

```
public class TestClass {
  void test1(int sym0, int sym1) {
    Absolute abs = new Absolute(sym0);
    abs.difference(sym1);
  }

  public static void main(String[] args) {
    TestClass tc = new TestClass();
    tc.test1(100, 1);
  }
}
```

In the driver, the integer inputs to constructor `Absolute` and method `difference` are transformed into the arguments of the `test1` test method. The `test1` method is called from the main method that is added as an entry point for dynamic symbolic execution. Finally, JDart is instructed that the `sym0` and `sym1` inputs to `test1` are treated symbolically.

3.3.3 Dynamic Symbolic Execution of Sequences

The last stage of every iteration is exploring the generated driver programs using dynamic symbolic execution as implemented in JDart. JDart explores paths through each driver program by solving path constraints over the specified symbolic inputs as described in Section 3.2.1. In the process, it generates additional unit tests, where each unit test corresponds to an explored path. The generated unit tests are added into the final set of unit tests. In addition to generating these unit tests, we also collect all the concrete input values that JDart generates in the process. We add these values back into Randoop's concrete primitive value pool for the sequence generation stage of the next iteration. By doing this, we feed the information that the dynamic symbolic execution generates back into the feedback-directed random testing stage.

3.4 Experimental Evaluation

We aim to answer the following research questions using the results of our empirical evaluation.

1. Can JDoop cover paths that plain random test case generation does not, and how big is the positive impact of covering such paths? To answer this question, we compare the performance of Randoop (as our baseline) and JDoop, using code coverage as a metric for the quality of the generated test suites.

2. Can dynamic symbolic execution enable randomized test case generation to access regions of a SUT that remain untested otherwise, i.e., does the feedback loop from JDart to Randoop (see Figure 3.5) have a measurable impact on achieved coverage? To answer this question, we run JDoop in multiple configurations with varying amounts of runtime attributed to Randoop and JDart, enabling a feedback loop in some configurations and preventing it in others.

3. What are the constituting factors impacting the effectiveness of JDoop in terms of the code coverage that can be achieved through automated generation of test suites? More specifically, can we confirm or refute the conjecture from related work [GFA13] that robustness of the used dynamic symbolic execution engine is pivotal or do other factors exist that have an impact on the achievable coverage (e.g., selection of test cases for symbolic execution)? To answer this question, we analyze statistics produced by JDart and vary the strategy in JDoop for selecting method sequences for execution with JDart as discussed in Section 3.3 (either selecting sequences randomly or prioritizing those with many symbolic variables).

In the remainder of this section, we introduce the benchmarks we used in our evaluation, describe our experimental setup, and present and discuss the results of the evaluation.

3.4.1 Benchmarks

We performed our empirical evaluation using the SF110 benchmark suite [sf113]. The suite consists of 110 Java projects that were randomly selected from the SourceForge repository of free software to reduce the threat to external validity (see Section 3.6). In our evaluation, we chose the largest subset of SF110 that both JDoop and Randoop can successfully execute on. Benchmarks that were excluded can be grouped into the following categories: unsuitable environment, inadequate or empty benchmarks, and deficiencies of testing tools. In the unsuitable environment category, benchmarks require privileged permissions in the operating system, a properly set configuration file, or a graphical subsystem to be available. There are several empty benchmarks, benchmarks that call the `System.exit()` method that is not trapped by testing tools, and benchmarks that are otherwise inadequate because of conflicting dependencies with our testing infrastructure. Finally, for some benchmarks, Randoop generates test cases that do not compile. All such problematic benchmarks were excluded from consideration, which left us with 41 benchmarks total, as listed in Table 3.1. For each benchmark, we list the number of instructions, branches, methods, and classes, which demonstrates we use a wide range of SUTs in terms of their size and complexity.

3.4.2 Experimental Setup

We used two tools in our empirical evaluation: JDoop and Randoop (version 3.0.10). For the comparison, we used Randoop 3.0.10, which is a version released after several months of interaction with the Randoop authors, during which we reported numerous issues and bugs that we found in the tool. We used JDoop 2.0 in the comparison, which is the latest version. In the evaluation, we explored several configurations of JDoop, where each configuration is determined by three parameters. The first parameter is the time limit for the first stage of every iteration, which is when Randoop runs (see Section 3.2.2); we vary this parameter as 1, 9, and 20 minutes. The second parameter is the time limit for the second and third stages combined, which is when JDart runs; we vary this parameter as 1, 9, and 40 minutes. The third parameter determines the strategy for selecting constructor+method call sequences as candidates for dynamic symbolic execution between: (1) random selection (denoted by 'R'), and (2) prioritization based on the number of symbolic variables (denoted

Table 3.1. Benchmarks from SF110 used in the evaluation.

Benchmark	Branches	Instructions	Methods	Classes
1_tullibee	915	8402	204	19
2_a4j	544	9773	522	45
3_gaj	22	415	52	10
5_templateit	564	5391	195	23
6_jnfe	132	7545	339	52
7_sfmis	146	4386	185	19
9_falselight	16	1189	32	14
11_imsmart	103	2244	86	17
13_jdbacl	3098	49385	1578	198
14_omjstate	52	954	67	14
16_templatedetails	38	656	87	24
22_byuic	2124	15031	195	14
23_jwbf	949	16032	609	86
26_jipa	128	1488	36	5
28_greencow	0	7	2	1
30_bpmail	208	3372	208	32
31_xisemele	150	3036	269	50
34_sbmlreader2	76	1447	26	8
37_petsoar	208	3445	377	58
42_asphodel	64	1139	101	20
46_nutzenportfolio	1183	18335	826	62
47_dvd-homevideo	376	10670	161	48
48_resources4j	312	3223	104	12
49_diebierse	197	4859	185	19
50_biff	814	7348	49	6
53_shp2kml	26	656	30	6
55_lavalamp	128	2907	236	48
63_objectexplorer	959	14118	902	84
65_gsftp	517	6587	181	32
67_gae-app-manager	68	1405	46	8
68_biblestudy	424	6005	313	23
69_lhamacaw	2016	51698	1437	101
72_battlecry	674	9550	130	15
74_fixsuite	374	6520	241	36
76_dash-framework	12	188	37	17
79_twfbplayer	1132	18315	902	160
84_ifx-framework	299	136363	26257	3900
90_dcparseargs	88	654	21	6
94_jclo	110	1094	43	4
95_celwars2009	850	15208	164	32
98_trans-locator	40	1097	39	6

by 'P'). Each configuration is code-named as JDoop-O-J-S, where O is the time limit for Randoop, J is the time limit for JDart, and S is the sequence selection strategy used. We explored the following six JDoop configurations: JDoop-1-9-P, JDoop-1-9-R, JDoop-9-1-P, JDoop-9-1-R, JDoop-20-40-P, and JDoop-20-40-R.

We carried out the evaluation in the Emulab testbed infrastructure [WLS+02]. We used 20 identical machines, each of which was equipped with two 2.4 GHz 64-bit 8-core processors, 64 GB of DDR4 RAM, and an SSD disk; the machines were running Ubuntu 16.04. We developed our testing infrastructure around the Apache Spark cluster computing framework. To facilitate reproducibility, each execution of a testing tool on a benchmark is performed in a pristine sandboxed virtualization environment. This is achieved via LXC containers running a reproducible build of Debian GNU/Linux code-named Stretch. We allocated four dedicated CPU cores and 8 GB of RAM to each container. Both Randoop and JDoop are multithreaded, and hence they utilized the multiple available CPU cores. Our testing infrastructure is freely available for others to use and extend.[2]

We allocate a 1-hour time limit per benchmark per testing tool/configuration for test case generation. Subsequent test case compilation and code coverage measurement phases are not counted toward the 1-hour time limit. Given that both Randoop and JDoop employ randomized heuristics, we repeat each run five times to account for this variability: for each benchmark we compute an average and a standard deviation. In terms of code coverage metrics, we measure instruction and branch coverage at the Java bytecode level using JaCoCo [jac17]. Furthermore, to get more insights into the performance of JDoop's symbolic execution engine (JDart), we collect statistics on the number of successful and failed runs, additional test cases it generates, symbolic variables in driver programs, times a constraint solver could not find valuation for a path condition, and JDart runs that explored one path versus multiple paths.

3.4.3 Evaluation of Test Coverage

Table 3.2 gives branch coverage and Table 3.3 gives instruction coverage results for each tool and configuration on all of the benchmarks. Figure 3.6 and Figure 3.7 provide the same results in a graphical form. Most results are stable across multiple runs, meaning that the calculated standard deviations are very small. In particular, the standard deviations for Randoop on a vast majority of benchmarks are 0, even though we used a different random seed for every run. This suggests that Randoop reaches saturation and is unable to cover more branches. For the most part, there are no major differences in terms of the achieved coverage between different tools/configurations. However, JDoop (in one of its configurations) consistently achieves higher coverage than Randoop. Given that pure Randoop saturates, we can conclude that the improvements in coverage we observe with JDoop are due to leveraging dynamic symbolic execution. Among JDoop configurations,

[2]The testing infrastructure is available under the GNU Affero GPLv3+ license at `https://github.com/soarlab/jdoop-wrapper`.

Table 3.2. Branch coverage results in percentage points. The results are averaged across five runs and including standard deviations. Highest coverage per benchmark is given in bold.

Benchmark	Randoop	1-9-P	1-9-R	9-1-P	9-1-R	20-40-P	20-40-R
1_tullibee	28.0 ± 1.2	29.4 ± 1.4	29.7 ± 1.0	29.4 ± 0.0	**31.6 ± 0.5**	28.7 ± 0.0	29.0 ± 0.2
2_a4j	58.5 ± 0.5	57.9 ± 0.0	60.0 ± 0.6	59.6 ± 0.2	**62.4 ± 0.1**	60.7 ± 0.1	60.7 ± 0.0
3_gaj	40.9 ± 0.0	40.9 ± 0.0	40.9 ± 0.0	40.9 ± 0.0	40.9 ± 0.0	40.9 ± 0.0	40.9 ± 0.0
5_templateit	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0
6_jnfe	48.5 ± 0.0	48.5 ± 0.0	48.5 ± 0.0	48.5 ± 0.0	48.5 ± 0.0	48.5 ± 0.0	48.5 ± 0.0
7_sfmis	35.9 ± 0.9	40.4 ± 4.2	39.7 ± 4.2	**42.5 ± 0.0**	40.5 ± 5.5	37.5 ± 2.7	37.1 ± 2.7
9_falselight	6.3 ± 0.0	6.3 ± 0.0	6.3 ± 0.0	6.3 ± 0.0	6.3 ± 0.0	6.3 ± 0.0	6.3 ± 0.0
11_imsmart	17.5 ± 0.0	17.5 ± 0.0	17.5 ± 0.0	17.5 ± 0.0	17.5 ± 0.0	17.5 ± 0.0	17.5 ± 0.0
13_jdbacl	36.6 ± 0.7	32.2 ± 3.1	32.2 ± 1.8	37.0 ± 0.5	**38.5 ± 0.6**	34.2 ± 1.0	33.6 ± 0.8
14_omjstate	48.1 ± 0.0	48.1 ± 0.0	48.1 ± 0.0	48.1 ± 0.0	**48.8 ± 3.1**	42.3 ± 0.0	42.3 ± 0.0
16_templatedetails	71.1 ± 0.0	68.4 ± 0.0	70.0 ± 1.8	71.1 ± 0.0	71.1 ± 0.0	68.4 ± 0.0	68.4 ± 0.0
22_byuic	7.8 ± 0.0	7.8 ± 0.0	7.8 ± 0.0	7.8 ± 0.0	7.8 ± 0.0	7.8 ± 0.0	7.8 ± 0.0
23_jwbf	26.6 ± 2.1	26.5 ± 1.7	27.2 ± 0.9	28.0 ± 0.6	**28.2 ± 1.9**	26.1 ± 0.5	26.0 ± 0.0
26_jipa	18.8 ± 0.0	24.2 ± 0.0	24.2 ± 0.0	24.2 ± 0.0	24.2 ± 0.0	23.4 ± 0.0	23.4 ± 0.0
28_greencow	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
30_bpmail	36.9 ± 0.5	36.9 ± 1.5	36.1 ± 1.2	**37.3 ± 0.6**	37.2 ± 0.6	37.2 ± 0.6	37.1 ± 0.5
31_xisemele	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
34_sbmlreader2	10.5 ± 0.0	10.5 ± 0.0	10.5 ± 0.0	10.5 ± 0.0	10.5 ± 0.0	10.5 ± 0.0	10.5 ± 0.0
37_petsoar	**54.1 ± 0.7**	52.8 ± 1.6	52.9 ± 1.4	53.4 ± 0.0	53.4 ± 0.0	53.7 ± 0.7	53.7 ± 0.7
42_asphodel	9.4 ± 0.0	9.4 ± 0.0	9.4 ± 0.0	9.4 ± 0.0	9.4 ± 0.0	9.4 ± 0.0	9.4 ± 0.0
46_nutzenportfolio	5.5 ± 0.0	5.2 ± 0.0	5.3 ± 1.6	5.6 ± 0.0	5.6 ± 0.6	5.5 ± 0.0	5.5 ± 0.0
47_dvd-homevideo	0.8 ± 0.0	0.8 ± 0.0	0.8 ± 0.0	0.8 ± 0.0	0.8 ± 0.0	0.8 ± 0.0	0.8 ± 0.0
48_resources4j	0.6 ± 0.0	0.6 ± 0.0	0.6 ± 0.0	0.6 ± 0.0	0.6 ± 0.0	0.6 ± 0.0	0.6 ± 0.0
49_diebierse	13.7 ± 0.0	13.4 ± 3.0	**19.7 ± 1.0**	14.2 ± 0.0	15.2 ± 15.1	13.7 ± 0.0	18.6 ± 13.1
50_biff	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0
53_shp2kml	19.2 ± 0.0	19.2 ± 0.0	19.2 ± 0.0	19.2 ± 0.0	19.2 ± 0.0	19.2 ± 0.0	19.2 ± 0.0
55_lavalamp	49.8 ± 0.6	48.4 ± 0.0	48.8 ± 1.6	51.9 ± 0.7	**52.0 ± 0.7**	48.4 ± 0.0	48.0 ± 2.0
63_objectexplorer	25.3 ± 0.0	24.6 ± 1.8	24.5 ± 1.0	**26.4 ± 0.3**	26.3 ± 0.9	25.0 ± 0.0	25.0 ± 0.2
65_gsftp	9.8 ± 1.0	9.9 ± 1.0	**10.0 ± 0.9**	9.9 ± 0.0	9.9 ± 0.0	9.5 ± 0.0	9.5 ± 0.0
67_gae-app-manager	2.9 ± 0.0	2.9 ± 0.0	2.9 ± 0.0	2.9 ± 0.0	2.9 ± 0.0	2.9 ± 0.0	2.9 ± 0.0
68_biblestudy	**37.5 ± 0.0**	36.9 ± 0.7	37.0 ± 0.4	37.3 ± 0.0	37.2 ± 0.8	37.0 ± 0.0	37.0 ± 0.0
69_lhamacaw	42.7 ± 0.4	40.1 ± 0.6	39.9 ± 1.0	**46.1 ± 0.5**	45.7 ± 0.6	40.3 ± 0.7	40.1 ± 0.4
72_battlecry	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0	0.1 ± 0.0
74_fixsuite	17.5 ± 6.5	17.1 ± 3.1	15.5 ± 1.3	19.2 ± 1.8	**19.6 ± 4.0**	17.4 ± 2.8	17.2 ± 3.3
76_dash-framework	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0	50.0 ± 0.0
79_twfbplayer	27.3 ± 0.0	23.2 ± 1.8	21.5 ± 1.3	29.4 ± 0.0	29.3 ± 1.0	**29.5 ± 0.0**	29.4 ± 0.1
84_ifx-framework	30.8 ± 0.0	32.6 ± 9.7	31.0 ± 8.9	**32.9 ± 2.8**	32.0 ± 2.8	29.5 ± 4.9	28.8 ± 2.3
90_dcparseargs	64.8 ± 0.0	64.8 ± 0.0	64.8 ± 0.0	64.8 ± 0.0	64.8 ± 0.0	64.8 ± 0.0	64.8 ± 0.0
94_jclo	42.7 ± 0.0	**46.0 ± 1.6**	44.5 ± 0.0	44.5 ± 0.0	44.7 ± 0.8	44.5 ± 0.0	44.5 ± 0.0
95_celwars2009	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0	**2.2 ± 5.2**	2.1 ± 0.0	2.1 ± 0.0	2.1 ± 0.0
98_trans-locator	25.0 ± 0.0	15.0 ± 36.5	18.0 ± 37.7	25.0 ± 0.0	**27.0 ± 3.7**	25.0 ± 0.0	25.0 ± 0.0

best-performing are the two 9-1 configurations where in an iteration Randoop runs for nine minutes and JDart for one minute; there are six such iterations in the 1-hour time limit.

Table 3.4 gives the total number of generated test cases. We do not observe a correlation between the number of generated test cases and achieved coverage. The JDoop-9-1-P configuration almost always generates the highest number of test cases. We conjecture that this is because more new concrete values are discovered by JDart than in the JDoop-9-1-R configuration (see Table 3.5), which leads to new test cases being generated faster by Randoop.

3.4.4 Profiling Dynamic Symbolic Execution

To analyze the potential impact of the robustness of dynamic symbolic execution on the validity of our results, we collected data from runs on all benchmarks for all configurations. We perform this analysis on data from single runs of JDoop as the other results show very

Table 3.3. Instruction coverage results in percentage points. The results are averaged across five runs and including standard deviations. Highest coverage per benchmark is given in bold.

Benchmark	**Randoop**	**1-9-P**	**1-9-R**	**9-1-P**	**9-1-R**	**20-40-P**	**20-40-R**
1_tullibee	42.1 ± 0.5	43.1 ± 0.2	43.0 ± 0.4	42.8 ± 0.0	**43.2 ± 0.2**	42.7 ± 0.0	42.7 ± 0.0
2_a4j	82.3 ± 0.4	83.2 ± 0.0	83.4 ± 0.2	83.0 ± 0.0	**84.3 ± 0.1**	83.1 ± 0.0	83.1 ± 0.0
3_gaj	59.3 ± 0.0	59.3 ± 0.0	59.3 ± 0.0	59.3 ± 0.0	59.3 ± 0.0	59.3 ± 0.0	59.3 ± 0.0
5_templateit	8.5 ± 0.0	8.5 ± 0.0	8.5 ± 0.0	8.5 ± 0.0	8.5 ± 0.0	8.5 ± 0.0	8.5 ± 0.0
6_jnfe	79.2 ± 0.0	79.2 ± 0.0	79.2 ± 0.0	79.2 ± 0.0	79.2 ± 0.0	79.2 ± 0.0	79.2 ± 0.0
7_sfmis	68.0 ± 0.1	69.5 ± 0.9	69.2 ± 1.0	**70.5 ± 0.0**	69.7 ± 1.5	68.4 ± 0.4	68.2 ± 0.4
9_falselight	39.8 ± 0.0	42.0 ± 0.0	42.0 ± 0.0	42.0 ± 0.0	42.0 ± 0.0	39.8 ± 0.0	39.8 ± 0.0
11_imsmart	33.9 ± 0.0	33.9 ± 0.0	33.9 ± 0.0	33.9 ± 0.0	33.9 ± 0.0	33.4 ± 0.0	33.4 ± 0.0
13_jdbacl	**52.1 ± 0.6**	47.4 ± 1.4	47.6 ± 1.7	50.9 ± 1.7	52.1 ± 1.2	49.5 ± 1.5	49.4 ± 2.0
14_omjstate	38.5 ± 0.0	38.5 ± 0.0	38.5 ± 0.0	38.5 ± 0.0	**38.9 ± 2.4**	37.2 ± 0.0	37.2 ± 0.0
16_templatedetails	89.3 ± 0.0	83.5 ± 0.0	86.0 ± 3.1	89.3 ± 0.0	89.3 ± 0.0	83.5 ± 0.0	83.5 ± 0.0
22_byuic	17.8 ± 0.0	17.8 ± 0.0	18.0 ± 0.0	17.8 ± 0.0	18.0 ± 0.0	17.8 ± 0.0	17.8 ± 0.0
23_jwbf	50.0 ± 0.4	50.1 ± 0.5	50.4 ± 0.3	50.7 ± 0.2	**50.8 ± 0.8**	49.7 ± 0.1	49.7 ± 0.0
26_jipa	50.8 ± 0.0	52.9 ± 0.0	52.9 ± 0.0	52.9 ± 0.0	52.9 ± 0.0	52.9 ± 0.0	52.9 ± 0.0
28_greencow	42.9 ± 0.0	42.9 ± 0.0	42.9 ± 0.0	42.9 ± 0.0	42.9 ± 0.0	42.9 ± 0.0	42.9 ± 0.0
30_bpmail	**42.3 ± 0.0**	42.3 ± 0.2	41.9 ± 0.7	42.3 ± 0.2	42.3 ± 0.1	42.2 ± 0.4	42.3 ± 0.1
31_xisemele	6.6 ± 0.0	6.6 ± 0.0	6.6 ± 0.0	6.6 ± 0.0	6.6 ± 0.0	6.6 ± 0.0	6.6 ± 0.0
34_sbmlreader2	11.0 ± 0.0	11.0 ± 0.0	11.0 ± 0.0	11.0 ± 0.0	11.0 ± 0.0	11.0 ± 0.0	11.0 ± 0.0
37_petsoar	63.5 ± 0.5	62.9 ± 1.0	62.9 ± 0.7	63.0 ± 0.1	63.0 ± 0.0	63.6 ± 0.5	63.6 ± 0.5
42_asphodel	26.4 ± 0.0	26.4 ± 0.0	26.4 ± 0.0	26.4 ± 0.0	26.4 ± 0.0	26.4 ± 0.0	26.4 ± 0.0
46_nutzenportfolio	16.6 ± 0.1	16.5 ± 0.1	16.5 ± 0.1	16.7 ± 0.0	16.7 ± 0.0	16.6 ± 0.0	16.6 ± 0.0
47_dvd-homevideo	2.0 ± 0.0	2.0 ± 0.0	2.0 ± 0.0	2.0 ± 0.0	2.0 ± 0.0	2.0 ± 0.0	2.0 ± 0.0
48_resources4j	3.7 ± 0.0	3.7 ± 0.0	3.7 ± 0.0	3.7 ± 0.0	3.7 ± 0.0	3.7 ± 0.0	3.7 ± 0.0
49_diebierse	23.3 ± 0.3	22.7 ± 0.8	23.4 ± 1.0	23.3 ± 0.0	23.4 ± 0.8	23.2 ± 0.0	**23.6 ± 0.8**
50_biff	3.5 ± 0.0	3.5 ± 0.0	3.5 ± 0.0	3.5 ± 0.0	3.5 ± 0.0	3.5 ± 0.0	3.5 ± 0.0
53_shp2kml	38.0 ± 0.0	38.0 ± 0.0	38.0 ± 0.0	38.0 ± 0.0	38.0 ± 0.0	38.0 ± 0.0	38.0 ± 0.0
55_lavalamp	64.8 ± 0.1	63.4 ± 0.0	63.7 ± 0.7	**65.2 ± 0.9**	65.0 ± 0.1	64.1 ± 0.0	64.1 ± 0.2
63_objectexplorer	27.3 ± 0.0	25.9 ± 1.8	26.0 ± 1.6	27.8 ± 0.2	**27.9 ± 0.4**	26.5 ± 0.1	26.6 ± 0.4
65_gsftp	10.1 ± 0.0	10.2 ± 0.0	10.2 ± 0.0	10.2 ± 0.0	10.2 ± 0.0	10.2 ± 0.0	10.2 ± 0.0
67_gae-app-manager	55.1 ± 0.0	55.1 ± 0.0	55.1 ± 0.0	55.1 ± 0.0	55.1 ± 0.0	55.1 ± 0.0	55.1 ± 0.0
68_biblestudy	**40.3 ± 0.0**	39.9 ± 0.4	40.0 ± 0.2	40.0 ± 0.0	40.1 ± 0.4	40.0 ± 0.0	40.0 ± 0.0
69_lhamacaw	41.1 ± 0.4	39.5 ± 0.5	39.5 ± 0.5	**42.4 ± 0.5**	42.2 ± 0.6	39.5 ± 0.9	39.3 ± 0.2
72_battlecry	0.5 ± 0.0	0.5 ± 0.0	0.5 ± 0.0	0.5 ± 0.0	0.5 ± 0.0	0.5 ± 0.0	0.5 ± 0.0
74_fixsuite	38.1 ± 11.2	42.8 ± 0.4	34.3 ± 0.2	**43.7 ± 0.2**	43.7 ± 0.5	43.0 ± 0.5	42.8 ± 0.4
76_dash-framework	66.0 ± 0.0	66.0 ± 0.0	66.0 ± 0.0	66.0 ± 0.0	66.0 ± 0.0	66.0 ± 0.0	66.0 ± 0.0
79_twfbplayer	33.9 ± 0.0	36.5 ± 0.6	35.9 ± 0.3	40.6 ± 0.0	40.6 ± 0.5	41.5 ± 0.0	41.5 ± 0.0
84_ifx-framework	91.1 ± 0.0	91.1 ± 0.1	91.1 ± 0.1	91.1 ± 0.0	91.1 ± 0.0	91.1 ± 0.0	91.0 ± 0.0
90_dcparseargs	55.0 ± 0.0	55.0 ± 0.0	55.0 ± 0.0	55.0 ± 0.0	55.0 ± 0.0	55.0 ± 0.0	55.0 ± 0.0
94_jclo	54.9 ± 0.0	**56.4 ± 0.7**	55.6 ± 0.0	55.6 ± 0.0	55.6 ± 0.0	55.6 ± 0.0	55.6 ± 0.0
95_celwars2009	3.3 ± 0.0	3.3 ± 0.0	3.3 ± 0.0	3.3 ± 0.4	3.3 ± 0.0	3.3 ± 0.0	3.3 ± 0.0
98_trans-locator	35.8 ± 0.0	33.2 ± 4.0	34.3 ± 4.8	35.8 ± 0.0	**36.2 ± 0.5**	35.8 ± 0.0	35.8 ± 0.0

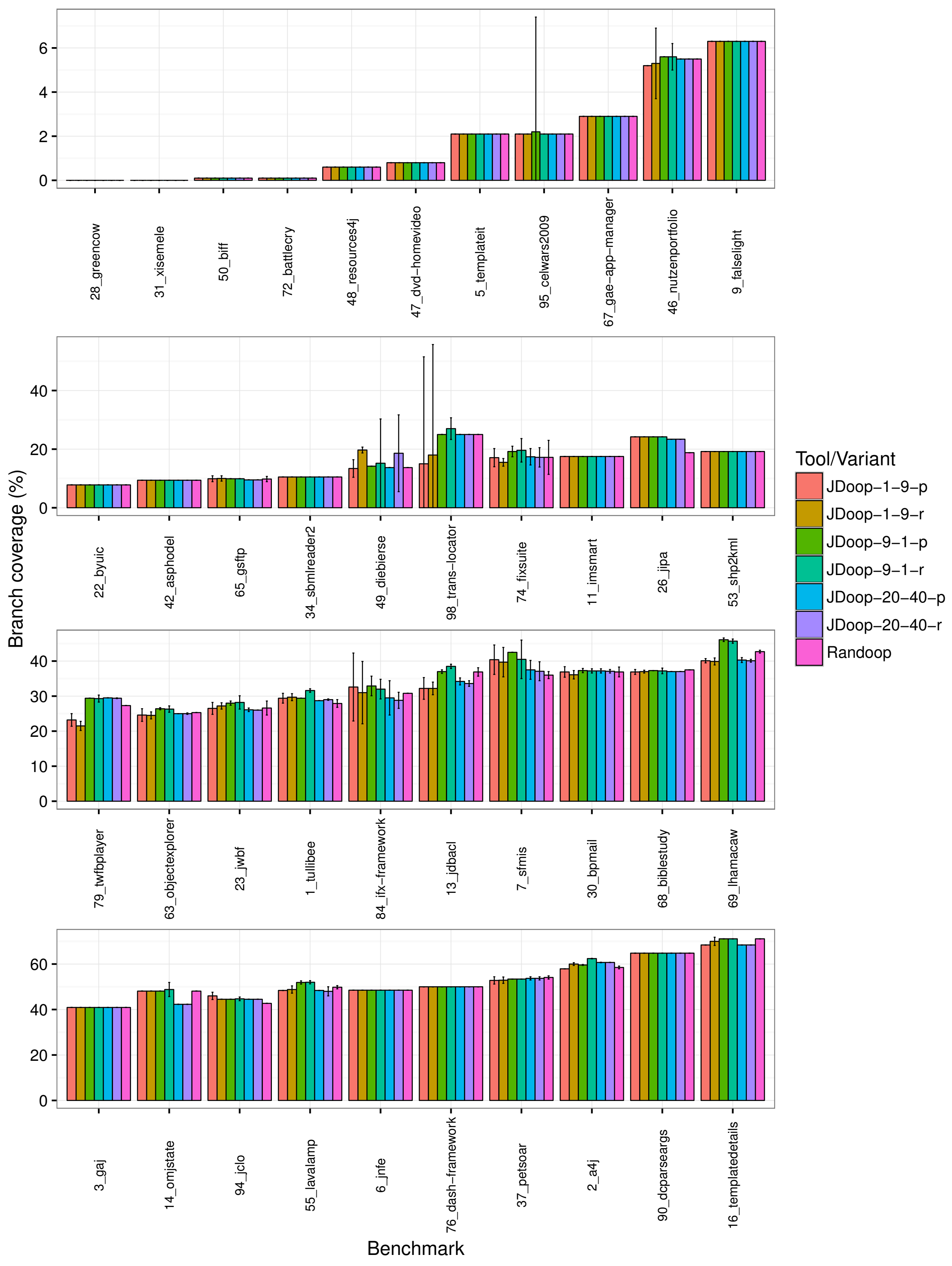

Figure 3.6. Branch coverage for RANDOOP and JDOOP. The results are averaged across five runs, where each tool and variant was given a time limit of 1 hour to generate test cases. Whiskers denote one standard deviation.

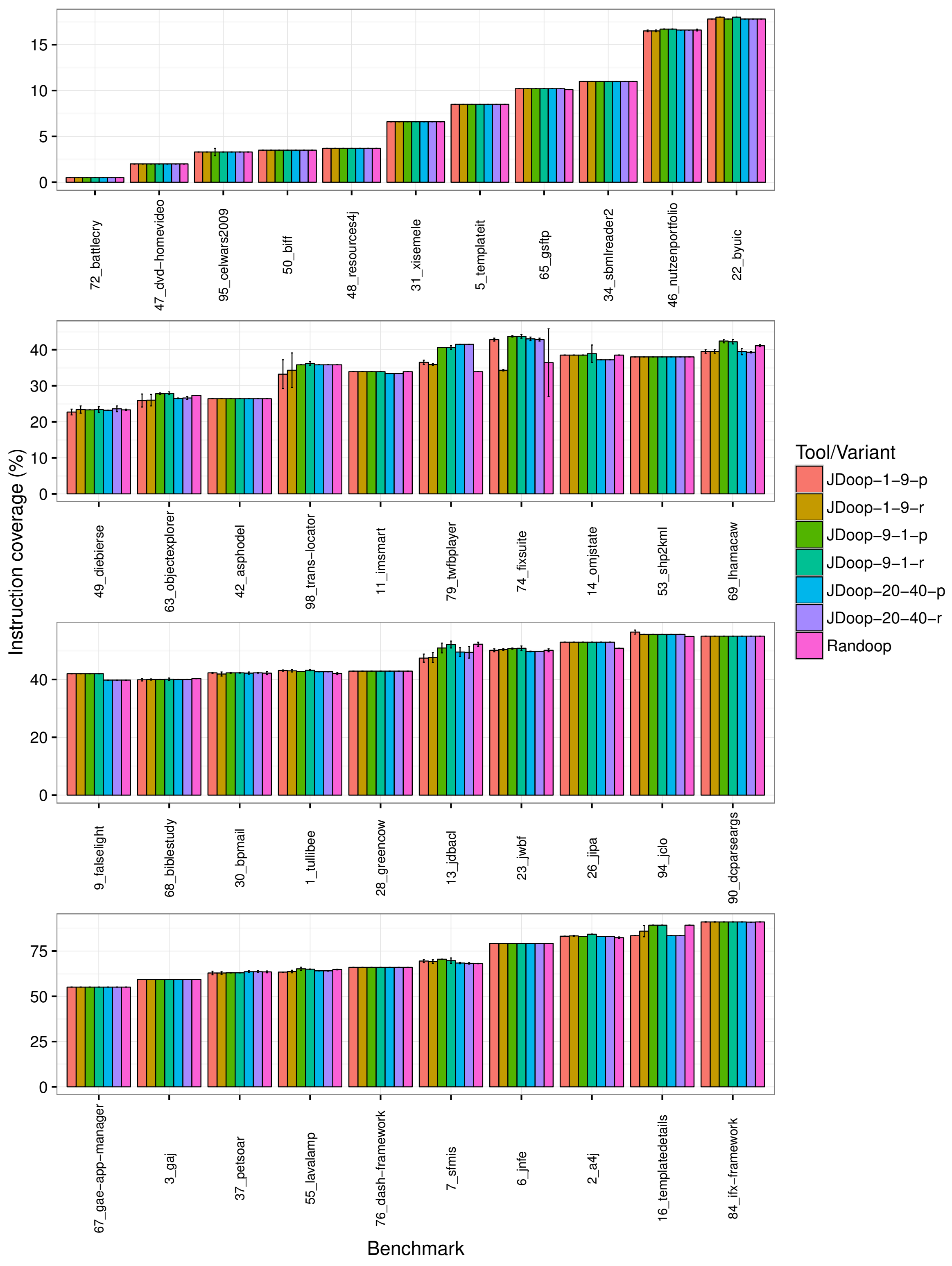

Figure 3.7. Instruction coverage for Randoop and JDoop. The results are averaged across five runs, where each tool and variant was given a time limit of 1 hour to generate test cases. Whiskers denote one standard deviation.

Table 3.4. Total number of generated test cases averaged across five runs. The highest and lowest number of test cases per benchmark are given in bold and italic, respectively.

Benchmark	Randoop	1-9-P	1-9-R	9-1-P	9-1-R	20-40-P	20-40-R
1_tullibee	327899	294360	218567	571372	**780346**	282531	*205016*
2_a4j	204778	*22430*	59123	57651	**329367**	104675	100769
3_gaj	92133	*40605*	29663	**189206**	124280	67709	45936
5_templateit	*2492*	**30552**	20002	29081	15570	6035	4930
6_jnfe	296501	91062	*81575*	411531	**423452**	119809	117743
7_sfmis	106590	106868	97875	**312046**	291029	76176	*68942*
9_falselight	253605	349925	185775	571283	**648156**	260382	*156277*
11_imsmart	92394	71420	70246	**139685**	134544	52892	*48136*
13_jdbacl	144769	54511	*40490*	213840	**224082**	100376	66342
14_omjstate	45258	19897	*15383*	**88114**	67370	36586	26753
16_templatedetails	85941	31274	*29697*	**180662**	147381	47189	42105
22_byuic	203736	171090	*95498*	**783132**	464633	231769	123310
23_jwbf	309571	101580	*83989*	**537897**	501119	190664	149524
26_jipa	10197	26398	17360	**56892**	30887	13684	*9424*
28_greencow	*2*	61	61	8	8	4	4
30_bpmail	192152	74257	*64353*	**351093**	325983	107954	91627
31_xisemele	169051	158073	138714	214357	**472635**	131312	*113848*
34_sbmlreader2	216223	98801	*83708*	**484943**	445669	149068	118315
37_petsoar	123934	44107	*36463*	**201709**	173353	71551	58942
42_asphodel	182354	77112	*65025*	**362966**	332233	114756	97165
46_nutzenportfolio	159105	38417	*27140*	**250170**	178543	127859	72222
47_dvd-homevideo	*4551*	51445	29280	**51645**	27500	8615	6189
48_resources4j	124493	46543	*42517*	**238827**	222356	77351	66348
49_diebierse	*14882*	65940	39903	**301354**	185464	74089	38992
50_biff	*88873*	**241906**	226403	162718	156676	128378	126911
53_shp2kml	159534	204167	118299	**575603**	404853	171718	*100513*
55_lavalamp	83663	20073	*19950*	**103958**	100395	57446	43075
63_objectexplorer	81502	57088	*41593*	**248669**	173910	70776	49294
65_gsftp	160055	122839	*68688*	**619504**	352394	178501	97830
67_gae-app-manager	8712	6492	6314	**11631**	8585	5666	*5477*
68_biblestudy	197945	66947	*41936*	**279601**	276622	143229	96125
69_lhamacaw	47097	26450	*19109*	**121782**	79115	47412	28249
72_battlecry	*63*	3143	3143	399	399	175	175
74_fixsuite	34597	33743	*23190*	**125728**	79302	41445	24942
76_dash-framework	*27430*	66864	**66977**	43307	43106	36546	33940
79_twfbplayer	*3818*	8083	6790	**14152**	8529	8930	6753
84_ifx-framework	*98087*	137964	136555	**215055**	213937	141820	136549
90_dcparseargs	100995	46582	*32062*	**245450**	159392	64925	43534
94_jclo	129468	67316	*50617*	163389	**244185**	85283	65468
95_celwars2009	172279	191373	*100317*	**751397**	409514	207416	105749
98_trans-locator	96104	78569	*44926*	**350256**	216182	110986	61545

little variation of results between different runs in most cases. Table 3.5 reports statistics on the JDart operation in different series of experiments. Data in the table are explained and discussed in the following paragraphs.

3.4.4.1 Modes of Operation

For all of the analyzed configurations of JDoop, JDart runs successfully in the vast majority of cases and produces significant numbers of test cases (up to $16,588$ in total for all benchmarks in one experiment). Most additional test cases are produced in the JDoop-1-9 configurations that enable the feedback loop between Randoop and JDart but grant the bulk of runtime to JDart. Across all configurations, random selection of method sequences for JDart leads to generating additional test cases for more benchmarks than prioritizing sequences with many symbolic variables. Prioritization, on the other hand, leads to more additional test cases in total.

Table 3.5. Statistics produced by JDART for single runs of all benchmarks in different configurations of JDOOP. JDART uses NHANDLER in the 'No Native' mode, except for one experiment that we performed in the 'Concrete Native' mode.

	JDOOP-20-40		JDOOP-1-9		JDOOP-9-1		JDOOP-9-1 (Concrete Native)
Sequence Selection Strategy	R	P	R	P	R	P	R
Potential Impact / Best Mode of Operation							
# Successful Runs	33,390.00	28,316.00	46,976.00	43,770.00	4,629.00	1,017.00	3,885.00
Successful Runs (%)	98.50	97.76	98.22	97.50	98.50	100.00	96.30
# Additional Tests	6,436.00	10,802.00	11,272.00	16,588.00	914.00	5,382.00	n/a
# Benchmarks with Additional Tests	19.00	9.00	20.0	13.00	18.00	4.00	n/a
Robustness and Scalability of JDART							
# Failed Runs	507.00	648.00	853.00	1,121.00	69.00	0.00	148.00
due to unhandled native code	3.00	1.00	14.00	6.00	1.00	0.00	10.00
due to classloading in SUT	504.00	647.00	839.00	1,115.00	68.00	0.00	138.00
Failed Runs (%)	1.50	2.24	1.78	2.50	1.50	0.00	3.70
# D/K Paths	17.00	192.00	170.00	84.00	5.00	0.00	26,915.00
D/K Paths (%)	0.26	1.78	1.46	0.51	0.01	0.00	93.62
Amenable Test Cases							
# Symbolic Variables per Test Case (Avg.)	2.10	6.60	1.88	4.67	1.99	6.21	1.86
# Runs of Single Paths	32,410.00	27,293.00	45,268.00	42,162.00	4,495.00	988.00	2,801.00
# Runs with Multiple Paths	980.00	1,023.00	1,708.00	1,608.00	134.00	29.00	1,084.00

3.4.4.2 Robustness and Scalability

Our data indicate that JDart is robust. Only a small number of runs fail (between 0.0% and 2.5%). Of these failures, only a tiny fraction is due to unhandled native code (less than 1%).[3] The vast majority of failed runs is caused by class-path issues in the benchmarks (more than 99%). There are only very few cases in which the constraint solver was not able to solve constraints of all paths in symbolic execution trees (between 0.0% and 1.75%).

Using Nhandler in the 'Concrete Native' mode leads to native calls being executed faithfully and to longer recorded path conditions, as discussed in Section 3.2.3.5. This yields constraints that are marked as not solvable ("don't know" or D/K for short) in 93.62% of all discovered paths in symbolic execution trees. This indicates the likelihood of JDart not being able to explore most of the paths that could be explored with proper symbolic treatment of native methods. Table 3.6 reports the number of occurrences for all encountered native methods in one run of JDoop. As can be seen from the data, the `charAt` method of the String class offers by far the greatest potential for improving on the number of explored paths. Note, however, that numbers in the table do not necessarily translate into the same number of additional paths as occurrences are counted along paths in trees and the same method call may appear on multiple paths.

3.4.4.3 Amenable Test Cases

The number of symbolic variables per test case behaves as expected: it increases when using prioritization of sequences with many variables. Prioritization, however, comes at a cost since there tends to be more runs of JDart in configurations that do not use

[3]These are methods for which Nhandler was not configured to take over execution, leading to a crash of JDart. We configured Nhandler to take care of all native methods of java.lang.String.

Table 3.6. Symbolic variables introduced by Nhandler in the 'Concrete Native' mode in a single run of JDoop-9-1.

Method	Occurrences
java.lang.String.charAt(I)C	2,157,258
java.lang.String.indexOf(I)I	430,951
java.lang.String.indexOf(II)I	18,199
java.lang.Character.isWhitespace(C)Z	63,723
java.lang.Character.isLetterOrDigit(C)Z	18,517
java.lang.Character.toLowerCase(C)C	16,506
java.lang.Math.min(II)I	2,800
java.lang.Float.floatToRawIntBits(F)I	81
sun.misc.Unsafe.compareAndSwapInt(Ljava/lang/Object;JII)Z	4,008

prioritization. For all benchmarks, a high number of runs yields only one path and hence no additional test cases. A considerable number of these runs may be attributed to using NHANDLER in the 'No Native' mode, thereby hiding branches by not executing native code. On the other hand, even in the experiment in which NHANDLER was used in the 'Concrete Native' mode, two thirds of all runs explored only a single path. This indicates that many method sequences that were selected for JDART simply do not branch on symbolic variables.

3.4.5 Discussion

The obtained results allow us to provide answers to our research questions.

3.4.5.1 Question 1: Covering More Paths

In terms of branch coverage, JDOOP outperforms RANDOOP on 44% of the benchmarks and there is a tie between RANDOOP and JDOOP on 46% of the benchmarks; see Table 3.2. (In instruction coverage, JDOOP outperforms RANDOOP in 29% of the benchmarks; see Table 3.3.) Measured in percentage points, the margins are relatively slim in many cases. There are, however, cases in which the achieved branch coverage is increased by 28%, resulting in an increase in code coverage by 5.4 percentage points (26_jipa). On 40% of the benchmarks, no variation can be seen in coverage between the two approaches. Together with the little variance that is observed between different runs, this indicates that RANDOOP in many cases reaches a state where coverage is (nearly) saturated. It makes sense that in such a scenario, JDOOP does not add many percentage points in code coverage. It merely adds coverage through those hard-to-hit corner cases.

3.4.5.2 Question 2: Reachable Regions

Our results indicate that the feedback loop has a positive impact. The JDOOP-9-1 configurations perform better than other configurations in most cases. Regarding the time distribution between RANDOOP and JDART, the picture is not as clear. There is a lot more variation in the margins of coverage increase (or decrease sometimes) for the configuration that grants most of the time to JDART. In one particularly amenable case, this results in coverage increase by 43% (from 13.7% to 19.7% for 49_diebierse).

3.4.5.3 Question 3: Robustness of Symbolic Execution

Here, we have to refute the conjecture that was made in related work [GFA13], namely that a robust dynamic symbolic execution engine can reap big increases in code coverage, or at least curb expectations about achievable coverage increases. Our experiments showed that JDart handles most benchmarks without many problems. Proper analysis of native code (especially for String methods) certainly has the potential to improve code coverage further, but the consistently high number of symbolic analyses that result in a single path (even in the control experiment) points to another important factor that contributes to small margins: the generated test cases simply do not allow to explore many new branches in most cases.

The experiments even indicate that it does not pay off to prioritize method sequences with many variables for JDart. Prioritization adds cost twice: once for analyzing test cases and then for exploring with many variables. Taking into account the observation from the first answer, that Randoop (almost) achieves saturation of coverage in 1 hour; this again indicates that in JDoop, corner cases are discovered by JDart. Covering more search space beats investigating the few locations more intensively in such a scenario.

3.5 Related Work

Here we present work related to our work.

3.5.1 Symbolic Execution

Dynamic symbolic execution [GKS05, SMA05] is a well-known technique implemented by many automatic testing tools (e.g., [CDE08, GLM12, TH08, SA06]). For example, SAGE [GLM12] is a white-box fuzzer based on dynamic symbolic execution. It has been routinely applied to large software systems, such as media players and image processors, where it has been successful in finding security bugs. Khurshid et al. [KPV03] extend symbolic execution to support dynamically allocated structures, preconditions, and concurrency.

Several symbolic execution tools specifically target Java bytecode programs. A number of them implement dynamic symbolic execution via Java bytecode instrumentation. JCute [SA06], the first dynamic symbolic execution engine for Java, uses Soot [soo17] for instrumentation and lp_solve for constraint solving. CATG [TZHS15] uses ASM [asm17] for instrumentation and CVC4 [DRK$^+$14] for constraint solving. Another dynamic symbolic execution engine, LCT [KLS$^+$11], supports distributed exploration; it uses Boolector and Yices for solving, but it does not have support for float and double primitive types. A drawback of instrumentation-based tools is that instrumentation at the time of class loading is confined to the system under test (SUT). For example, LCT does not by default instrument the standard Java libraries, thus limiting symbolic execution only to the SUT classes. Hence, the instrumentation-based tools discussed above provide the possibility of using symbolic (and/or simplified) models for non-instrumented classes or using preinstrumented core Java classes.

Several dynamic symbolic execution tools for Java are not based on instrumentation. For example, the dynamic symbolic white-box fuzzer jFuzz [JHG09] is based on JPF (as is JDart) and can thus explore core Java classes without any prerequisites. Symbolic PathFinder (SPF) [PMB$^+$08] is a JPF extension similar to JDart. In fact, JDart reuses some of the core components of an older version of SPF, notably the solver interface and its implementations. While at its core SPF implements symbolic execution, it can also switch to concrete values in the spirit of dynamic symbolic execution [PRV11]. That enables it to deal with limitations of constraint solvers (e.g., nonlinear constraints).

3.5.2 Hybrid Approaches

There are several approaches similar to ours that combine fuzzing or a similar testing technique with dynamic symbolic execution. Garg et al. [GIB$^+$13] propose a combination of feedback-directed random testing and dynamic symbolic execution for C and C++ programs. However, they are addressing challenges of a different target language and on a much smaller collection of benchmarks that they simplified before evaluation. The Driller tool [SGS$^+$16] interleaves fuzzing and dynamic symbolic execution for bug finding in program binaries, and it targets single-file binaries in search of security bugs. Galeotti et al. [GFA13] apply dynamic symbolic execution in the EvoSuite tool to explore test cases generated with a genetic algorithm. Even though their evaluation is carried out in a different way than the one presented in this chapter, the general conclusion is the same in spirit: dynamic symbolic execution does not provide a lot of additional coverage on real-world object-oriented Java software on top of a random-based test case generation technique. MACE [CBP$^+$11] combines automata learning with dynamic symbolic execution to find security vulnerabilities in protocol implementations.

There are automated hybrid software testing tools that do not strictly combine with symbolic execution (e.g., OCAT [JKXC10], Agitator [BDS06], Evacon [IX08], Seeker [TXT$^+$11], DSD-Crasher [CSX08]). Because these tools either focus on a single method at a time or just form random method call sequences, they often fail to drive program execution to hard-to-reach sites in the SUT, which can result in suboptimal code coverage.

3.5.3 Random Testing

Randoop [PLEB07] is a feedback-directed random testing algorithm that forms random test cases that are sequences of method calls, while ensuring basic properties such as reflexivity, symmetry, and transitivity. Search-based software testing [McM11] approaches and tools are gaining traction, which is reflected in four annual search-based software testing tool competitions in recent years [RJGV16]. A prominent search-based tool is EvoSuite [FA11], which combines a genetic algorithm and dynamic symbolic execution. T3 [Pra16] is a randomized tool that generates constructor and method call sequences based on an optimization function. JTExpert [SPG16] keeps track of methods that can change the underlying object and constructs method sequences that are likely to get the object into a desired state. All the search-based testing tools are geared toward testing at the class level, while JDoop performs testing at the application/library level.

3.5.4 Benchmarking Infrastructures

In computer science, any extensive empirical evaluation, software competition, or reproducible research requires a significant software+hardware infrastructure. The Software Verification Competition's BenchExec [Bey16] is a software infrastructure for evaluating verification tools on programs containing properties to verify. It comes with an interface for verification tools to follow, which did not fit our needs: our coverage measurement outcomes cannot be judged in terms of program correctness. The Search-based Software Testing Competition [RJGV16] community created an infrastructure for the competition as well. However, just like tools that participate in the competition, their infrastructure is geared toward running a testing tool on just one class at a time. Emulab [WLS$^+$02] and Apt [RWS$^+$15] are testbeds that provide researchers with an accessible hardware and software infrastructure. They allow for repeatable and reproducible research, especially in the domain of computer systems, by providing an environment to specify used hardware, on top of which users can install and configure a variety of systems.

3.6 Threats to Validity

In this section, we cover threats to validity of herein presented work.

3.6.1 Threats to External Validity

While the main purpose of the SF110 corpus of benchmarks is to reduce the threat to external validity since they were chosen randomly, we cannot be absolutely sure that the benchmarks we used are representative of Java programs. Hence, our results might not generalize to all programs. In JDoop, we combined Randoop and JDart, and we used Randoop as the baseline in our evaluation. We attempted to include another testing tool into the comparison, in particular EvoSuite. However, to the best of our ability, we did not manage to get it to work with JaCoCo (the tool we use for measuring code coverage) despite being in contact with the EvoSuite authors; hence, we could not perform a direct comparison and our results might not generalize to other tools. Having said that, earlier work on EvoSuite reports similar results to ours with respect to using dynamic symbolic execution in combination with random testing [GFA13]. Finally, note that we do not include the environment and dependencies of benchmarks into unit test generation, which might lead to sub-optimal coverage in some cases.

3.6.2 Threats to Internal Validity

In our evaluation, we experimented with 3 different time allocations for Randoop and JDart that we identified as representative. While our results show no major differences between these different time allocations, we did not fully explore this space and there might be a ratio that would lead to a different outcome. JDart currently cannot symbolically explore native calls, which might lead to not being able to cover program paths (and hence also branches and instructions) that depend on such calls. Our evaluation shows that this indeed happens and that native implementations of methods of the String class in Java are the main culprit, but it does not allow us to provide an estimate for the impact on the achieved code coverage. Finally, while we extensively tested JDoop to make sure it is reliable and performed sanity checks of our results, there is a chance for a bug to have crept in that would influence our results.

3.6.3 Threats to Construct Validity

Here, the main threat is the metrics we used to assess the quality of the generated test suites, and in particular branch coverage. This threat is reduced by previous work showing that branch coverage performs well as a criterion for comparing test suites [GGZ+13].

3.7 Conclusion

We introduced a hybrid automatic testing approach for object-oriented software, described its implementation JDoop, and performed an extensive empirical exploration of this space. Our approach is a combination of feedback-directed random testing (Randoop) and dynamic symbolic execution (JDart), where random testing performs global exploration, while dynamic symbolic execution performs local exploration (around interesting global states) of the SUT. It is an iterative algorithm where these two exploration techniques are interleaved in multiple iterations. Our evaluation on real-world object-oriented software shows that dynamic symbolic execution provides modest, albeit consistent, improvements in terms of code coverage on top of our baseline (pure feedback-directed random testing).

CHAPTER 4

AUTOMATIC TESTING FOR RUNTIME VERIFICATION

This chapter is based on work published at the International Symposium on Software Testing and Analysis 2015 [DG15].[1]

4.1 Introduction

The Next Generation Air Transportation System (NextGen) is a NASA research program that addresses the increasing load on the air traffic control system through innovative algorithms and software systems. AutoResolver is a proposed NextGen component for prediction and resolution of loss of separation between multiple aircraft in the one to eight minutes time horizon. Loss of separation between two aircraft occurs when they are closer to each other than a predefined safe vertical or horizontal distance. Separation assurance aims to eliminate the occurrence of loss of separation in the air space. Figure 4.1 shows a sketch of a potential loss of separation between two aircraft and how it can be avoided by letting one aircraft take a detour.

Testing AutoResolver presents various challenges. The input data consists of several aircraft trajectories, each trajectory being a sequence of 4-dimensional points, where a point represents a position in 3-dimensional space, plus a time instant. Given this complex input

[1]Portions of the published work are reused and reprinted here with permission.

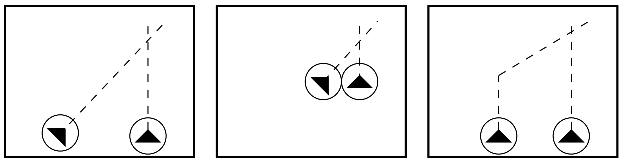

Figure 4.1. Loss of separation and resolution. Lateral view of two aircraft, their trajectories, and areas of horizontal separation assurance. Left: Conflict in the near future. Center: Loss of separation. Right: Detour that will prevent loss of separation.

space of the separation assurance problem, it is extremely hard to generate appropriate input data for it. Therefore, the NextGen team typically uses historical airport data recordings as test inputs. Each such test case usually involves (tens of) thousands of aircraft and takes several hours to run. When unexpected behavior is detected, it is hard to create subsets of the test case that would lead AutoResolver to similar behavior, making debugging a complicated task.

To address these issues, the Robust Software Engineering (RSE) and NextGen groups at the NASA Ames Research Center have been collaborating over several years for the development of an automated, lightweight testing infrastructure for AutoResolver. In previous work [GHI+14], Giannakopoulou et al. developed a wrapper that implements parameterized loss of separation scenarios between two aircraft for AutoResolver. In contrast to having trajectories as inputs, which would make it impossible for test-case generation tools to produce realistic trajectories, these scenarios expose parameters such as aircraft velocity and heading. These parameters provide flexibility in producing many different types of aircraft encounters for the problem, while always producing valid trajectory inputs that also exhibit loss of separation. Despite the fact that scenarios are limited to a single encounter between two aircraft, the wrapper has enabled them to experiment with both black-box and white-box test-case generation techniques, and to produce hundreds of thousands of tests aiming at exercising different aspects of the AutoResolver code.

A key aspect missing from their work [GHI+14] is the identification of desirable properties for separation assurance software, and the support for automated testing of these properties. Introduced properties may, in turn, create additional requirements on the test-case generation itself: to exercise each property, we often need to produce complex encounter situations that target it. In general, it is hard to identify properties for separation assurance algorithms, as discussed in TSafe [GBS+11]. Currently, AutoResolver developers manually examine the outcomes of every test case to determine whether the software behaves as expected. This is impractical, more so in a setting where millions of test cases are generated and run automatically.

The work presented in this chapter addresses these issues, thus filling an important gap in our separation assurance testing framework. Specifically, it makes the following contributions:

- A set of requirements for separation assurance.
- Some of the requirements require test cases with multiple aircraft, including complex relationships between their trajectories. We therefore implement a generalization of the wrapper to support scenarios with any number of aircraft and loss of separation cases between them. Some aircraft are placed strategically in order to check specific properties of AutoResolver's logic.
- A runtime verification framework based on aspect-oriented programming for checking the requirements on AutoResolver. Runtime monitoring is also used to check property coverage by the generated tests, as well as to monitor other behaviors of the system, as requested by its developers. The framework is completely separate from the AutoResolver code, thus allowing us to avoid interfering with the development process of the AutoResolver team.
- As well as being used for property monitoring, our runtime verification framework is used for complex test-case generation. This is, to our knowledge, a novel, atypical use of runtime verification.
- Application of our framework to AutoResolver and discussion of the obtained results.

The remainder of this chapter is organized as follows. Section 4.2 provides preliminaries to AutoResolver and a previous testing framework for it. Separation assurance requirements for a system like AutoResolver are given in Section 4.3. Section 4.4 presents extensions to the interface and to the test-case generation capabilities of our testing framework, with Section 4.5 describing the runtime verification infrastructure that we develop on top of it. Evaluation results follow in Section 4.6, and lessons learned are provided in Section 4.7. Finally, related work and conclusions are discussed in Section 4.8 and Section 4.9, respectively.

4.2 Preliminaries

This section provides a high-level description of AutoResolver and provides a brief overview of previous work on developing a testing environment for it. Note that in the context of this work, we use the terms "conflict" and "loss of separation" interchangeably. Moreover, by the term "conflict time," or "*ttlos*," we refer to the first time point in their trajectories at which two aircraft lose separation.

4.2.1 AutoResolver

The Advanced Airspace Concept (AAC) is aimed at automating separation assurance in the future. AAC uses multiple independent layers of separation assurance for increased reliability. One component of AAC is a strategic problem-solving tool named AutoResolver [ELC10]. AutoResolver's separation assurance algorithm was originally developed in the ACES simulation environment taking full advantage of the zero-error trajectory prediction available. Many studies of the effectiveness of this algorithm in the zero-uncertainty environment have been performed [FE07].

In each round of operation, AutoResolver attempts to resolve all the conflicts identified by its conflict-detection system, but handles them in the order specified by some notion of priority. The algorithm that it implements attempts to generate many different types of resolutions for each conflict. More specifically, AutoResolver iteratively attempts a variety of maneuvers, and determines which ones result in successful resolutions. Among all such resolution trajectories, AutoResolver selects the resolution that is expected to impart the minimum airborne delay.

AutoResolver consists of approximately $65K$ lines of Java code. For each maneuver that it attempts, it communicates with ACES, a simulation environment whose core consists of approximately $450K$ lines of code. As mentioned, the NextGen team typically uses historical airport data recordings as test inputs. These test cases consist of $4,800$ or $10,000$ aircraft, and take between three and seven hours to run. The results are monitored manually by domain experts to see whether AutoResolver behaves as expected.

4.2.2 Lightweight Testing Framework

To ensure more systematic testing of the behavior of AutoResolver, potentially targeting some type of test coverage, Giannakopoulou et al. previously developed a lighter-weight testing environment to complement the current testing process [GHI$^+$14]. That testing environment has the following features. Despite the fact that ACES is very precise, it is a heavyweight tool that adds a significant burden to the testing process. Giannakopoulou et al.

therefore created stubs that replace the functionality of ACES with more approximate behavior. The main capability that the ACES stubs provide is the generation and modification of aircraft trajectories. Stubbing ACES allowed them to run tests significantly faster, which is important in a setting where millions of test cases are generated and executed automatically.

Moreover, to enable meaningful test-case generation, Giannakopoulou et al. created a modular, extensible wrapper around AutoResolver that implements parameterized conflict scenarios. Note that loss of separation is handled for two aircraft at a time (all other aircraft are called "secondary"). Each proposed resolution is evaluated against the set of all aircraft in the airspace sector that AutoResolver is currently handling.

More precisely, the wrapper provides an interface to AutoResolver that consists of a number of entry points where each point represents a single-conflict scenario suggested by the AutoResolver team. In a Cruise scenario, each aircraft is in level flight, i.e., its altitude remains constant. In a Climb scenario, each aircraft climbs or descends for some portion of the trajectory. Loss of separation may occur before one or both of them start climbing or descending, during climb or descent, or after one or both of them has leveled off. Finally, the Turn scenario is similar to Cruise, but it introduces a heading change for one or both aircraft at some point in their trajectory. Each scenario is hard-coded in the type of trajectories, but is parameterized on aspects like aircraft velocity, initial heading, initial altitude, climb rate, and conflict time. Each concretized scenario is translated into a set of trajectories with which the wrapper invokes AutoResolver.

In order to ensure that the test inputs that are thus generated mostly correspond to loss of separation scenarios, the framework works as follows. A point is selected in 3-dimensional space, representing a position at which the two aircraft will meet (note that loss of separation actually occurs prior to the airplanes reaching that point). Aircraft are then flown backwards (headings are reversed) from their meeting point for the duration specified by the conflict time parameter. The type of trajectory is as specified by the scenario, i.e., whether the aircraft climb or cruise, for example, but specific details are parameterized, such as the time point at which a climb trajectory levels off. Aircraft initial positions are thus computed, and the ACES stub is then invoked to generate concrete trajectories.

The wrapper enables the application of a variety of test-case generation tools or algorithms for this problem, as demonstrated in previous work. In this work, the focus is on an essential aspect of testing, which is how to evaluate the testing outcomes. In particular, we want to lighten the load of domain experts in monitoring each test case. Rather, our goal is to automatically check each test case against requirements, and focus the attention of the domain experts only on those tests that exhibit unexpected behavior.

4.3 Separation Assurance Requirements

The first step in verifying the behavior of AutoResolver is to create a specification, i.e., a set of requirements it should meet, which is not straightforward for a system that solves an optimization problem. Through several iterations of discussions with the AutoResolver team, we have formed two types of requirements: verification properties and information monitors. Verification properties are statements about the expected behavior of AutoResolver. They capture the high-level logic of the system, which conveys how the system operates in various situations. Information monitors provide a rich insight into specifics of the logic, which are not necessarily characterized as correct or incorrect.

4.3.1 Verification Properties

P_1: *There should be a resolution for every conflict.* The main goal of AutoResolver is to predict and resolve conflicts. Therefore, it should be able to resolve every conflict that occurs within its time horizon.

P_2: *Initial conflicts are resolved in the non-decreasing order of their time to first loss of separation.* It is important that AutoResolver handles conflicts with earlier conflict time first. Initial conflicts are those detected before a conflict resolution process starts. Conflicts that will happen in, e.g., seven minutes are not as urgent to resolve as those that will happen in one minute. In this way, the conflict resolution process prioritizes to resolve more imminent conflicts first, and move to later conflicts according to their time to loss of separation.

P_3: *New conflicts arising as a result of conflict resolution should be inserted into the list of conflicts according to their time to first loss of separation.* As AutoResolver is resolving initial conflicts, new conflicts can be created as a result of the resolution process. This means that a resolution trajectory of the maneuvered aircraft has a conflict with another aircraft, while its original trajectory did not have a conflict with the same aircraft. When

AutoResolver picks a resolution that results in a new conflict, the new conflict should be resolved according to its conflict time. This property is similar to property P_2, but instead it characterizes conflicts that did not exist originally.

P_4: *No picked resolution is allowed to cause a more imminent secondary conflict.* For every conflict AutoResolver tries to resolve, it attempts multiple resolutions. Among the successful ones, it picks the best according to a set of optimization criteria. The picked resolution should not make the situation worse by getting the aircraft into a more imminent conflict than the conflict being resolved.

4.3.2 Information Monitors

The AutoResolver team is interested in the stability of the picked resolution's type for a given conflict if the resolution process is delayed. Each resolution has a type, such as a horizontal maneuver, a temporary altitude change maneuver, etc. The team said that stability is important from the perspective of people working in air traffic control. In other words, the team wants to know if AutoResolver would pick a different resolution type for the same conflict, if the resolution process was to be delayed for a given amount of time. We therefore formulate the following monitor:

M_1: *For each conflict, report its resolution type and how it changes over time.* We explain how we introduce a resolution delay in Section 4.4.

4.4 Extending AutoResolver's Testing Framework

Previous work [GHI+14] supported single-conflict preselected scenarios, in spite of AutoResolver being able to resolve any number of arbitrary conflicts at a time. Our extensions to the previous work are motivated by the need to support automated verification of separation assurance properties, and in particular the requirements presented in Section 4.3.

With this work, we aim to verify properties of AutoResolver's operational logic pertaining to more than one conflict. We therefore rewrite the framework interface to support the generation of any combination of any number of conflicting aircraft, including aircraft that create secondary conflicts (we call these secondary aircraft). An overview of the extended framework's architecture is provided in Figure 4.2.

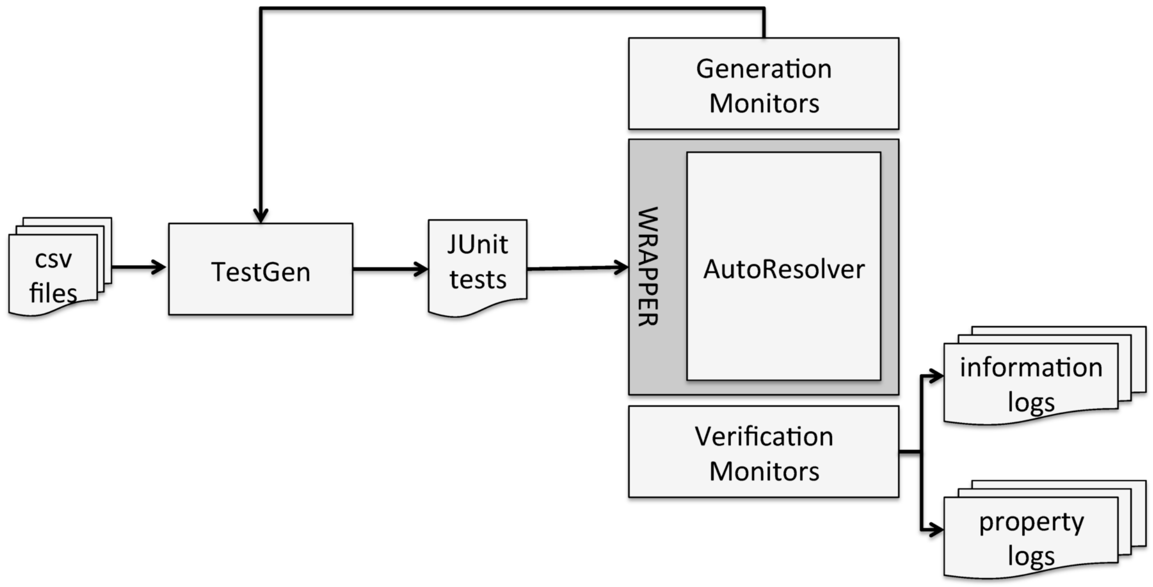

Figure 4.2. Overview of extended testing framework.

4.4.1 Framework Interface

The new interface decouples the task of creating an aircraft trajectory from that of creating a conflict encounter. More specifically, a single aircraft can be added that flies a parameterized type of trajectory (e.g., cruise or climb) through an arbitrary 4-dimensional point.[2] In other words, the trajectory is such that the aircraft is at the 3-dimensional location specified by this point, at the time that is associated with the point. Two aircraft are put in conflict by making them fly through the same 4-dimensional conflict point, or have them be in a secondary conflict, as explained in Section 4.4.3. Unlike the previous framework, encounter scenarios are not hard-coded, but are generated through the combination of aircraft with appropriate trajectories. For example, a Climb conflict is generated by flying two climbing aircraft to the same conflict point.

A simplified example of a test case we generate according to the new interface is shown in Figure 4.3. It features two pairs of aircraft resulting in two independent conflicts. Each aircraft is added to the framework by utilizing a setup interface entry point that is specific to a particular aircraft trajectory type. (For example, a Cruise trajectory is generated by calling `setUpCR`.) Each of the setup methods takes a number of parameters defining the

[2]One of the parameters enables a time offset to the time dimension of the specified point.

```
public void test0() throws Throwable {
  AacTestWrapper wrapper = new AacTestWrapper();

  wrapper.setUpCR(CR_params1, conflict_point1);
  wrapper.setUpCRH(CRH_params1, conflict_point1);
  wrapper.setUpCL(CL_params1, conflict_point2);
  wrapper.setUpCL(CL_params2, conflict_point2);

  wrapper.runConflictDetectionResolution();
}
```

Figure 4.3. A test case with two pairs of aircraft in two independent conflicts.

trajectory, and the targeted conflict point. When all aircraft are added to the framework, a wrapper call to AutoResolver's detection and resolution process is made with the last statement in the test case.

A new addition to the interface is the ability to fly all aircraft for a given amount of time before invoking AutoResolver's conflict detection and resolution process. With this ability, we stress-test AutoResolver's conflict detection and resolution of the same aircraft trajectories, but at different time points. At the same time, it allows us to observe how resolution types change over time. Once all aircraft are added to the framework, some of them exhibiting loss of separation, the framework invokes AutoResolver to detect and resolve conflicts between the aircraft.

4.4.2 Generating Multiple Conflicts

With the exception of P_1, all the properties reason about cases where AutoResolver handles multiple conflicts between more than two aircraft. In this work, we introduce a way of generating test cases with multiple independent conflicts. This enables us to exercise the properties and report potential violations.

In their previous work [GHI+14], Giannakopolou et al. generated test cases by computing the Cartesian product of all input parameters for an interface entry point describing a pre-selected single-conflict scenario. For the case of multiple conflicts, if we were to additionally compute the Cartesian product across conflict points, the resulting number of test cases would be prohibitively large. There are many ways in which we could handle this problem: one would be to implement a randomized approach to creating combinations of values; another would be to use combinatorial testing [CDPP96].

As a first approach to the problem, we decided to introduce initial conflicts at points in the 3-dimensional space that are far enough from each other such that two aircraft involved in a conflict do not interfere with aircraft from the other conflicts. By making the conflicts isolated from each other, characteristics of one conflict are expected to be independent from characteristics of every other conflict. Based on this, it is sufficient to explore the Cartesian product of the parameter values for each conflict, but it is not necessary to also explore the Cartesian product across conflicts.

Each initial conflict is described by several parameters such as the time to the first loss of separation, aircraft trajectories, relative headings of the two aircraft, etc. Each conflict parameter is a floating point or an integer value. We range over its values by specifying an interval with a lower bound, an upper bound, and a step between neighboring values to be extracted from the interval. We draw parameter specifications from input files, one line specifying one parameter, as illustrated in the following:

```
ttLOS,           DOUBLE, "[400.0 to 540.0 step 20.0]"
relativeHeading, DOUBLE, "[20.0  to 180.0 step 20.0]"
airspeed,        DOUBLE, "[450.0 to 550.0 step 50.0]"
```

For each conflict, we then explore the full Cartesian product of the ranges of all the parameters involved. However, we do not explore all combinations of parameters across conflicts. Rather, we create test cases by picking one set of generated values for each conflict, and covering all possible combinations of parameter values for each conflict. During this process, we avoid having the same time to loss of separation among conflicts in each test case by using different offsets in the order in which their respective values are picked.

4.4.3 Generating Secondary Conflicts

Properties P_3 and P_4 require for us to generate not simply additional aircraft, but also to strategically position them so that they generate secondary conflicts. In particular, we must create test cases that include secondary aircraft with which a resolved trajectory creates more and less imminent conflicts. If we were to take a random approach where we would include secondary aircraft without a specific goal, it would be extremely difficult to generate such test cases. We therefore decided to take control over positioning the secondary aircraft, in the same way that our wrapper takes control over generating initial conflicts.

The main challenge in generating secondary conflicts is the fact that we do not, a priori, know the resolution that AutoResolver will produce for a particular conflict. To deal with this problem, we develop the following approach, as illustrated in Figure 4.4. Let T_1 and T_2 be the trajectories of two aircraft in conflict, and assume that AutoResolver produces a resolution for T_1.

1. Monitor the behavior of AutoResolver and obtain the produced resolution. Generate a resolution trajectory T_1'.
2. Select a point in T_1' to which the secondary aircraft flies (black dot in Figure 4.4). Use existing algorithms to fly the secondary aircraft backwards to the selected point and generate a trajectory T_3. The secondary aircraft will thus be in conflict with the resolution trajectory produced by AutoResolver.
3. Create a test case with T_1, T_2, T_3.

Similarly to test case generation of initial aircraft encounters, we want to parameterize the generation of secondary aircraft in order to be able to create interesting test cases. For example, we want to vary the time at which the secondary conflict occurs. For a desired time t to a secondary conflict, we pick an appropriate point in the resolution trajectory (the point at which the aircraft will be at time t), and create a secondary aircraft to reach that point also in time t, with the method discussed in Section 4.2.2. This enables us to create secondary conflicts that are both more and less imminent than the initial conflict. We also want to vary heading and velocity of the secondary aircraft, among other parameters. Finally, we need to make sure that when AutoResolver deals with the generated test case, it will still select T_1 and T_2 as the first conflict to resolve. In other words, if the secondary aircraft is in conflict with any of the initial aircraft, and this conflict is more imminent

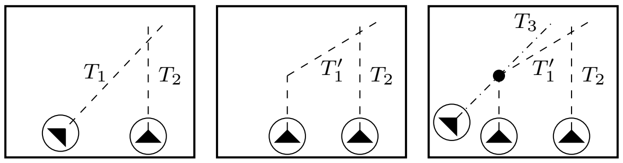

Figure 4.4. Loss of separation resolution and introduced secondary conflict. Left: conflict in the near future. Center: resolution that will prevent loss of separation. Right: secondary conflict introduced along the resolution trajectory.

than the conflict between T_1 and T_2, then AutoResolver will select the former conflict to resolver first, and therefore we no longer have control over the creation of the resolution trajectory with which we create a conflict.

To address this last issue, we use a runtime monitor that, prior to generating the test case including a secondary, checks whether the secondary aircraft has a conflict with the initial trajectories, and only generates a test case if the latter conflict is less imminent than the initial conflict. In other words, a test case such as above is only generated if the conflict between T_1 and T_2 is more imminent than both 1) a conflict between T_1 and T_3, if such a conflict exists, and 2) a conflict between T_2 and T_3, if it exists.

In summary, we use runtime monitors to query the behavior of AutoResolver and generate complex secondary scenarios in a strategic fashion. In this novel use of runtime monitoring for test-case generation, the software under test serves as a solver of our constraints for producing the test cases.

4.5 Verification and Monitoring

A new component of the framework was developed to support verification and monitoring capabilities: monitoring the execution of AutoResolver, checking if properties hold, and recording property violations and other information to appropriate log files.

Log files for property violations let the AutoResolver team focus on those test cases that violate the properties, instead of manually examining results of millions of test cases. These log files provide information analogous to a regression test suite: if there is a test case violating any of the properties, a report will be written to the log files. Log files for the information monitor report detailed information on every aircraft conflict for every test case and how the picked conflict resolution's type changes over time.

4.5.1 Monitoring Information and Properties

One of the goals of this work was to keep AutoResolver's source code intact throughout the verification. What is usually done when verifying a software is to instrument its source code with inlined commands needed for a verification task. The motivation behind avoiding such an invasive approach is the fact that we did not want to interfere with the ongoing development of AutoResolver.

Instead of modifying AUTORESOLVER at the source code level, we added a verification component to our framework. The AUTORESOLVER software is written in the JAVA programming language. JAVA compiles to JAVA bytecode, an intermediate language of the JAVA Virtual Machine. The verification component is written in the ASPECTJ language. ASPECTJ is an aspect-oriented programming extension to JAVA. It comes with a modified JAVA compiler that weaves the code of the component into AUTORESOLVER's bytecode. This way we have AUTORESOLVER's code interleaved with our verification code at the bytecode level only, thus leaving AUTORESOLVER's source code unmodified.

For every property and the monitor introduced in Section 4.3, we have one aspect in the component. An aspect is an analog of a JAVA class. It consists of regular JAVA code in addition to pointcuts and advices. Pointcuts are moments in a program execution, e.g., an occurrence of a method call. Advices are actions to be taken before and/or after the pointcuts, e.g., fetching a return value after a method call.

In order to implement the runtime verification, we had to identify parts of AUTORESOLVER's source code pertaining to the properties and monitor. Once the parts were identified, we were able to write aspects — consisting of pointcuts and advices — that implement the properties and monitor. The pointcuts also reach into the framework since the aspects initialize various parameters on the boundary from the framework to AUTORESOLVER. Furthermore, the aspects reach into test cases exercising the framework and AUTORESOLVER, again initializing parameters before a test case starts or outputting results after a test case execution or when a whole test suite ends with its execution. An example of a pointcut and an advice applied before the pointcut is shown in Listing 4.1.

Every aspect is instantiated at the beginning of a framework execution and it monitors an execution of each test case in a test suite, independently of other aspects.

```
pointcut executionJUnitTestMethod():
  execution(public void test*()) &&
  within(TestCase+) &&
  !cflow(myAspect());

before(): executionJUnitTestMethod() {
  currentTime = 0.0;
  resolutionInfoMap =
    new HashMap<AircraftIDPair, ArrayList<ResolutionInfo>>();
}
```

Listing 4.1. A pointcut and advice initializing parameters before a JUnit test case method execution.

Some information the aspect observes and collects is written to log files after every test case, like the `resolutionInfoMap` data structure in Listing 4.1, while the rest of the information is written to the files only at the end of a test suite execution.

The resolution monitor modifies each JUnit test case and executes it at nine different time points. The resulting executions represent a class of test cases with the exact same aircraft trajectories. However, for each delay time, all aircraft involved in the test case are first flown for the specified amount of time prior to invoking the AutoResolver's conflict detection and resolution process. This is equivalent to generating and executing nine times as many test cases as in the original JUnit test suite. A pointcut and advice that achieve the runtime modification and execution are shown in Listing 4.2.

In testing AutoResolver, we made it easy to focus on specific properties or the runtime monitor through a configuration file. The file contains key-value pairs, where a key represents one of the properties or the monitor, and a value is either enabled or disabled, allowing or disallowing the property or the monitor to be exercised, respectively. Note that when the information monitor is enabled, each test case results in multiple invocations to AutoResolver, one for each resolution delay that is introduced. In addition to that, all enabled property monitors are also active and checked as AutoResolver is invoked. For each test case, we therefore checked all enabled properties at all resolution delay time points.

4.5.2 Monitoring the Monitors

In their previous work [GHI+14], Giannakopoulou et al. measured and compared coverage of generated test suites in terms of both standard structural coverage criteria, but also in terms of the sets of attempted and successful resolutions.

```
pointcut callAR(AacTestWrapper wrapper):
  call(public ArrayList runConflictDetectionResolution()) &&
  target(wrapper) &&
  !cflow(myAspect()) &&
  !cflow(callFlyForMethod(*, *)) &&
  if(isEnabled);

after(AacTestWrapper wrapper): callAR(wrapper) {
  for (t = 60.0; t <= 480.0; t += 60.0) {
    AacTestWrapper w = wrapper.flyFor(t);
    w.runConflictDetectionResolution();
  }
}
```

Listing 4.2. A pointcut and advice delaying the conflict detection and resolution process for every JUnit test case.

When properties are introduced into the system, we have the opportunity to introduce additional coverage criteria for the generated test suites. In particular, we wished to make sure that our test suites include enough different scenarios to exercise the logic of all the properties that are to be checked in the system. One can view this as a type of "property coverage" criterion. If no test case exercises a particular property, our testing process satisfies this property "vacuously."

Let us consider property P_4, for example. This property requires that a picked resolution does not create a more imminent secondary conflict. One can imagine this property as having two branches, one for the case where a secondary conflict is created as a result of a resolution trajectory, and one where it is not. The first branch consists of two branches, one where the secondary conflict is more imminent than the original one, and one where it is less imminent. To cover this property, we therefore would like to have test cases that cover all three logical branches.

Similarly, property P_3 checks whether secondary conflicts are added to the list of conflicts in the correct order. In order to check the logic of this property, we must generate test cases that create secondary conflicts *within* the AutoResolver time horizon, so that they are eligible to be added to the list of conflicts.

To evaluate the quality of our generated test suites with respect to a set of properties, we introduced runtime monitors, which in essence observed whether the property monitors are exercised properly. This can be performed by extending the property monitor itself. For example, for property P_3, we extended the property monitor to additionally record the number of calls made to the method that adds secondary conflicts to an existing list of conflicts. Alternatively, it can be performed by introducing a new monitor. Since the lifespan of aspects is through a test suite, it is easy to aggregate such test-suite coverage results.

4.6 Experimental Evaluation

In this section, we report results on verifying the properties and running the monitor introduced in Section 4.3. We analyze if each of the properties holds and what insights this gives us into how AutoResolver operates. For the monitor, we look at the data it outputs and what it means.

4.6.1 Experimental Setup

Experiments were run on a 32-core computer with 128 GB of RAM in the Debian GNU/Linux operating system in the Emulab network testbed [WLS$^+$02]. The experiments consist of generating test cases for AutoResolver, as described in Section 4.4.2, and then checking the properties and running the monitor while executing the test cases, as explained in Section 4.5. Every JUnit test case we generate consists of five independent initial conflicts, which equals to five pairs of aircraft, each pair flying to a conflict point. In addition, there is another aircraft in the test case intended to cause a secondary conflict if AutoResolver were to choose the same conflict resolution for one of the conflicts in the presence of the additional aircraft.

It took 2.08 seconds on average to execute a test case. Given that it takes so long to execute a test case and that we generated 3.5 million test cases in total, we decided to implement the producer-consumer problem in the framework. The producer is the test case generator where each consumption unit is a batch of 5000 test cases. In the experiments, we employed 30 consumers, each consumer executing one batch at a time. With the producer-consumer approach, it took us about three days to run the test cases compared to about 84 days it would take us if we executed them sequentially.

One thing to note is that, as explained in Section 4.5, every test case is modified and executed in nine different ways at runtime. Effectively, this means we executed not 3.5 million test cases, but 31.5 million modified test cases. Thanks to the runtime modifications of every test case representing multiple conflict scenarios, we were able to check all the properties for the same scenarios, but at different time points.

4.6.2 Property Checking / System Monitoring

In this section, we present the results we obtained from the runtime monitors that we implemented as oracles during the execution of our generated tests.

P_1: *There should be a resolution for every conflict.* Monitoring of this property identified several test cases for which AutoResolver was not able to produce a resolution. All these cases fall within four categories, as reported by the output of AutoResolver. Three categories have to do with the conflict time. More precisely, there are some conflicts that AutoResolver considers outside (before or after) its time horizon. For really imminent conflicts, tools such as TSafe [GBS+11] and TCAS [KĐ07] are in charge. For conflicts that are far enough in the future (how far in the future is configurable), it is often better to wait and see how they evolve before trying to resolve them. After all, each aircraft maneuver that is to be applied is disruptive in one way or another. There are also cases identified as "planes already in violation," which means that the initial states of the trajectories are already in loss of separation.

The fourth category produces a message that we could not directly interpret based on our prior experience with the tool: "Neither plane able to maneuver/neither plane able to be unfrozen." The AutoResolver developers informed us that this happens when aircraft involved in a conflict have already received a resolution in this round. By reviewing the logs from these test cases, we confirmed that this was indeed the case. This is the type of behavior that we could not observe with the single-conflict test cases of our previous work.

The behavior of AutoResolver in all these cases is therefore as expected. Relating to the conflicts that fall outside of the AutoResolver time horizon, one could refine the property monitor to exclude such cases from being reported. Alternatively, to keep the property general, one could simply create filters to be applied offline to the logged test cases. One could also create parameterized runtime monitors that allow to configure the time window within which the target separation assurance algorithm is expected to operate.

P_2: *Initial conflicts are resolved in the nondecreasing order of their first time to loss of separation.* We did not detect any violations of this property.

P_3: *New conflicts arising as a result of conflict resolution should be inserted into the list of conflicts according to their first loss of separation time.* We did not detect any violations of this property.

P_4: *No picked resolution is allowed to cause a more imminent secondary conflict.* Our framework initially reported several violations of this property, with the following message, for example:

Resolution for conflict: (ac1=1, ac2=2) ttlos = 75.0 [s], res type = 13 is causing a more imminent conflict with ac3=3 with ttlos = 0.0 [s]

We observed that in all the cases reported, the more imminent secondary conflict that occurs as a result of applying a resolution maneuver occurs immediately, with *ttlos* equal to 0.0. We performed several tests to confirm that our runtime monitor was detecting a real violation, and that our runtime monitor does not have a bug. In trying to understand these violations with the AutoResolver team, we provided to them detailed trajectories involved in one of these test cases. One of the developers observed the fact that in our test cases, the initial point of the original trajectory did not coincide with the initial point of its corresponding resolution trajectory.

This fact exposed a bug in our wrapper, which we had not discovered previously. In debugging the issue, we noticed that when our ACES stub creates a trajectory, the first point that it adds to the trajectory is the first point to which the aircraft flies. In other words, all the trajectories generated by our stub are offset by five seconds (trajectory points are five seconds away from each other). This is not a serious problem since it does not really matter what the initial point of an aircraft is, for the purpose of our testing. However, when the ACES stub is asked to create a trajectory that starts at a particular initial point I, it is reasonable to expect that the first trajectory point will coincide with I.

Despite this fact, we were surprised that AutoResolver produced a resolution with a more imminent secondary conflict. We were expecting that when a resolution is attempted, AutoResolver would check whether it creates more imminent conflicts. Since the conflict detection algorithm detects such a conflict, how is it possible that AutoResolver selects the resolution? After several interactions with the AutoResolver developers on this issue, we realized that the logic of the algorithms used assumes that the initial point of original and resolution trajectories are the same. When we updated the ACES stub, the violations to this property disappeared.

This attests to the correctness of our runtime monitor. Runtime verification thus proves invaluable in debugging and understanding a system under test, which is impossible without the use of test oracles to assist in this process. We identified an issue with our ACES stub, but also discovered an implicit assumption made by the logic of AutoResolver. We believe that this feedback was useful to the AutoResolver team. An assumption is made that the trajectory generator will always create trajectories with a specific initial point, but why not robustify the logic to work correctly if a trajectory generator does not exactly satisfy this criterion?

M_1: *For each conflict, report its resolution type and how it changes over time.* This monitor produces, for each conflict of each test case, a report that looks as in Table 4.1.

Table 4.1. Resolution delay and resulting resolution types. Nonresolved conflict 1 is reported as being "before AutoResolver horizon," and 2 reports "planes already in violation." Resolution types are represented as integers in AutoResolver.

ttlos [s]	Delay time [s]	Res. type
430.0	0.0	26
370.0	60.0	26
310.0	120.0	26
250.0	180.0	26
190.0	240.0	26
130.0	300.0	13
70.0	360.0	13
10.0	420.0	not resolved[1]
0.0	480.0	not resolved[2]

This report allows for an easy inspection of how the produced resolution type changes when conflict resolution is postponed. Visual inspection is of course not the aim of our work. Monitor M_1 produces about 20GB of logged data. This information opens up many opportunities for data analysis. One could, for example, try to calculate the average, or minimum, or maximum delay at which the produced resolution type changes. One could also look for characteristics of outlier cases. Through observation of the reports, for example, we noticed some cases that look irregular; for example, it was sometimes the case that a conflict ahead disappears and then reappears between two aircraft, which is not very intuitive. In other cases, the resolution type changed very early, where in most cases, the resolution type only changed when we delay by at least a couple of minutes.

4.6.3 Property Coverage

As discussed, in addition to checking properties, we introduced monitors that check whether properties are exercised appropriately by the executed test suite. Our test suite exercised all properties appropriately except for property P_3. We were puzzled by this behavior. For this reason, we additionally monitored whether we generate test cases with the following characteristic: a resolution is picked that introduces a less imminent conflict than the original one, and the secondary conflict is within the AutoResolver horizon. After confirming this fact, we had extensive discussions with the AutoResolver team, and discovered that for the types of conflicts that we are implementing (called en-route), secondary conflicts are not handled in the current iteration of AutoResolver. Rather, they are left to be detected in the next invocation of the tool. Only weather conflicts, which we had not yet incorporated in our framework, may exercise this behavior.

4.7 Lessons Learned

Testing without oracles is a shot in the dark. The NextGen team and we thought that through the years of trying to understand how to generate test cases and stubs for a complex system like AutoResolver, we had learned many aspects of its behavior. However, the pace at which we discover new aspects of its logic has increased significantly with the introduction of runtime monitors. Monitoring the system has helped us identify assumptions of the AutoResolver logic, categories of unresolved conflicts that we were unaware of, as well as bugs in our own wrapper code (after all, the wrapper is a part of our system under test) that Giannakopoulou et al. had not previously discovered despite extensive testing [GHI$^+$14]. This is without counting all the information that the AutoResolver team expects to learn from the recorded data by the information monitor.

Strategic test-case generation and property monitoring are valuable tools in the hands of developers. In presenting our results to the wider separation assurance team, there were two things that sparked their enthusiasm. The first was the novel way in which we generate secondary aircraft. Of our other test-case generation work, they said that "we have done somewhat similar work for our own basic testing, although not nearly as advanced or flexible." However, they said that "we have never seen anything remotely similar to this way of generating secondary conflicts; it is very novel and interesting."

The second aspect of our work that they believed can make a real impact in the way they test their software is the runtime monitoring. They believed that verification through monitoring can significantly facilitate their testing and regression processes. They also appreciated the fact that monitors do not interfere with their development.

Properties trigger the creation of new properties. From this, but also others' past experience with TSafe [GBS$^+$11], it is clear that the best way to identify properties of complicated algorithms is to start from simple ones, and show the value to system developers. The capability to create logs of tests that identify several characteristics of interest immediately creates a desire in the team to monitor additional aspects of the system. This is a great opportunity for formal methods experts to get their hands on real systems.

Of course, it comes at the price of a significant engineering effort, which, in the NextGen team's case, has spanned over several years. However, the potential impact of our work makes the effort worthwhile and rewarding.

Our information monitor also opens up opportunities for applying a variety of data analysis and mining techniques in this domain. One example of properties that we have not yet explored are properties that compare resolutions across consecutive rounds of operation of AutoResolver. The difference from our current information monitor is that instead of flying aircraft in order to delay the resolution process, we would actually apply the picked resolutions, and invoke AutoResolver again on the resolved trajectories, at the next time point according to the frequency of invocation. We are inspired to specify such properties from work on analyzing the ACAS X system [vEG14]. In that work, an example of undesirable behavior is described as "reversals," which describes a situation where the same aircraft is advised to climb and subsequently to descend. Such maneuvers are extremely disruptive to the pilot.

Runtime monitoring is useful for test-case generation. In this work, we used runtime monitoring in conventional ways, to check properties of the system under test, but also in innovative ways, to generate test cases that are hard to generate with other techniques. In particular, there are two such interesting examples. The first one has to do with creating secondary conflicts. In this example, the system under test is used as a "constraint solver"; it informs our test-case generation tool of the resolution that AutoResolver would pick for a specific conflict. We believe that such an approach could be applicable in many real-world scenarios where we are required to solve constraints that are very particular to a specific application.

The second case is where the runtime monitor captures information from one test case, modifies it, and subsequently runs the modified test case, as performed by our information monitor. M_1 modifies each test case at runtime to effectively generate a whole class of related test cases with the same aircraft setup, but across time.

4.8 Related Work

In the research field of runtime verification, researchers have explored a number of formalisms, usually with a trade-off between expressiveness and performance. Java PathExplorer [HR04] checks an execution of a Java program against user-provided properties and analyzes the program for deadlocks and data races. LogFire [Hav14] is a rule-based runtime verification system based on a pattern-matching algorithm for implementing production rule systems. JavaMOP [CR07] enables the user to specify properties using a formalism, and automatically generates monitors in AspectJ from the specification. Even though we could have used a tool such as JavaMOP in this work, we did not need its expressiveness, hence we encoded the requirements in AspectJ directly. For an overview of the runtime verification field, see Leucker and Schallhart [LS09]. Runtime verification is also used in software-fault monitoring; Delgado et al. [DGR04] provide a taxonomy and a classification of software-fault monitoring systems.

A combination of test-case generation and runtime verification has been explored by others working on runtime verification. The jUnit^{RV} tool [DLT13] extends the JUnit unit testing framework with annotations that are generated from a user-specified temporal logic formula. The tool manipulates Java bytecode at runtime. Artho et al. [ABG$^+$05] use an input-output model to automatically generate test cases enriched with verification properties. The test cases are generated using symbolic execution and the properties are analyzed by applying runtime verification to execution traces. In their approach, the system under test is instrumented manually so that events of interest are recorded in the execution traces. In our work, runtime verification is instrumental in test-case generation, and instrumentation is done automatically at the Java bytecode level, without affecting the source code of the software under test. Furthermore, in our work, runtime monitors and test cases are separate, offering the flexibility to apply any selection of runtime monitors to any selection of test cases (whether generated automatically or manually).

The oracle problem of identifying correct output, which we faced in this line of work, is also approached by metamorphic testing [ZHT$^+$04]. Chen et al. [CCY98] show how to augment passing test cases in metamorphic testing with the goal of revealing undetected errors in software.

The AutoResolver system has previously been integrated and evaluated with other National Airspace System (NAS) simulations [MT08, PHM$^+$09, Thi08]. Moreover, in previous work, Giannakopoulou et al. tested TSafe [GBS$^+$11], a NextGen component that also targets separation assurance, but at a shorter time horizon. That work also identifies properties to be tested, but these properties involve the input and output data of the system, and therefore do not require more involved monitoring, as in this work. Moreover, the input space of TSafe is much smaller than that of AutoResolver, making test-case generation simpler.

Several researchers have addressed the challenge of generating structurally complex inputs with white- and black-box techniques. Techniques range from using declarative specifications of the test inputs [BKM02, GGJ$^+$10] to white-box techniques based on concolic execution for security testing [GLM12]. MACE [CBP$^+$11] combines black- and white-box techniques such as active automata learning and concolic execution in order to increase code coverage. A lot of research combines the techniques to generate method sequences and input values for primitive parameter types and objects [TH08, IX08, TXT$^+$11, DR13, PLEB07, GIB$^+$13]. Arnold and Alexander [AA13] generate complex tests based on an automated approach to generating content for computer games. Other researchers rely on program invariants inferred from executions in generating test cases [BDS06, CSX08].

As mentioned, other test-case generation approaches can be added to our framework such as plain random input generation, concolic execution, combinatorial testing, and evolutionary test case generation [PHP99]. The use of such techniques is only made possible by our implementation of a wrapper to tame the input space of AutoResolver.

4.9 Conclusion

The presented work here is based on several years of work with experts in aircraft separation assurance to develop a lightweight testing environment for the AutoResolver tool. In this work, we particularly focused on specifying and monitoring properties of the AutoResolver algorithms during testing. We discussed how we implemented property and information monitors in the AspectJ language, allowing us to log several aspects of the system without interfering with its source code.

To effectively exercise properties of interest of the system, we had to generate sophisticated test cases. To this aim, we used runtime monitoring in innovative ways, both as an oracle for constraint solving, and as a generator of classes of related test cases.

Note that in this work, we did not focus on experimenting with different test-case generation tools. Rather, we focused on creating appropriate interfaces around AutoResolver, which tame its input space of trajectories into parameterized scenarios that can be handled by test-case generation tools. Here we used a simple black-box test-case generation algorithm.

We automatically generated and efficiently executed millions of test cases, and discovered errors and vulnerabilities in the system consisting of AutoResolver and our wrapper code. We also gained new insights in the logic of the algorithms. The separation assurance team at NASA Ames has expressed interest in incorporating our work more widely. One possible way to extend the framework is in creating generic monitor templates for separation assurance, that can be configured to work with a variety of tools and algorithms in that domain.

Finally, the AutoResolver team would like us to stress-test the system by placing conflict points closer together, for example, and thus having aircraft from different conflicts interfere with each other. They are also interested in classifying input tests for which the AutoResolver may not be able to produce a resolution. In general, the information we produce opens many opportunities for further analysis.

Even though we have so far focused our work on air-traffic control algorithms, the approaches that we have developed are relevant in other domains. For example, techniques described here could be applied in the car industry to test algorithms for self-driving cars.

CHAPTER 5

CONCLUSION

This concludes the dissertation. In it, we presented three lines of work revolving around automatic software testing. All three lines extended and applied several testing techniques in novel ways with the goal of increasing software reliability.

In Chapter 2 on automatic testing in malware detection, we analyzed how automatic software techniques can be used to build high-quality detectors of malicious software behavior. In particular, we looked at how the type and size of input in automatic random testing affects four quality measures in various machine learning-based binary classifiers. We applied this analysis to thousands of applications for the Android operating system. Results showed that the classifiers can be built efficiently based on simple inputs guiding application execution. The quality of the classifiers was affected only negligibly when varying the number of inputs from several hundred to several thousand.

In the first stage of this work, we were planning to employ dynamic symbolic execution in driving application execution. However, there was no ready-to-use dynamic symbolic execution engine for Android applications; hence, we decided to start with random testing and first see results this approach gives. We were pleasantly surprised by preliminary results and good quality detectors, which made us revisit both our original plan with symbolic execution and earlier work of others on malware detection with heavyweight techniques.

Detectors we have built are robust and not dependent on particular applications, as shown by our analysis. They provide insights into how different configurations in testing influence detectors. Random testing was shown to be effective and did not leave much room for improvement that more advanced and complex techniques can bring.

With Chapter 3, we introduced a technique for automatic test case generation for object-oriented software. As explained in the chapter, the challenge was in generating objects of arbitrary complex classes in the subtyping polymorphism sense. The motivation behind this work is in the fact that covering more code with automatic testing triggers more faulty sites in a software code base. Therefore, we needed to cover as much of code full of objects, where each object is generated by either following an unspecified protocol or by setting up an uncovered state in which an object can be generated.

There we combined two well-known existing automatic testing techniques. The combination consists of feedback-directed random testing and dynamic symbolic execution, forming a novel technique that eliminates drawbacks of both underlying techniques. Random testing provided us with call sequences that broadly explored execution states, while symbolic execution enabled us to explore the neighborhood of a state in detail. Our extensive evaluation of the technique on a scale that has not been undertaken before showed on a wide range of benchmarks that the combined technique provides modest improvements in code coverage over the feedback-directed random testing technique.

Chapter 4 addressed the challenge of automatically testing and checking properties of an optimization system. It is difficult to capture correctness of states in an optimization system as computed states in it are compared in relative terms to each other, and not against an unknown optimal state. There we proposed an automatic software testing technique for NASA's AutoResolver aircraft collision avoidance system. First we formulated four verification properties and one information property. The properties we formulated gave us clarity, i.e., something we could work with in terms of testing and verification.

In a novel way, our technique interleaved a black-box test case generation approach with runtime analysis and verification to 1) guide the iterative construction of complex test cases that stress-tested the system, and 2) check if the four properties hold for in-flight scenarios given in the automatically generated test cases. The technique enabled us to use the system under test in an interesting way by both putting it into a position of a test case generator and an oracle to correctness of simpler scenarios that were consequently used in constructing complex aircraft scenarios. Test cases we generated were generic and without properties baked in. Rather, properties were specified in and checked on the fly by the verification framework, thereby decoupling test case generation and property specification and verification.

Our approach in testing the aircraft collision avoidance system led to discovery of errors and vulnerabilities in the system comprising AutoResolver and our testing framework.

With the work, we gained new insights in the logic of the algorithms in the AutoResolver system. The AutoResolver team's airspace engineers were happy to be provided with an automatic test case generation and verification framework for AutoResolver. This significantly improved over their manual testing process. The framework enabled them to develop new features and test their behavior more rapidly and with higher confidence of correctness.

With these three chapters of the dissertation, we demonstrated our thesis that automatic software testing can be combined with machine learning, dynamic symbolic execution, and runtime verification to broaden its applicability.

REFERENCES

[AA13] James Arnold and Rob Alexander. Testing Autonomous Robot Control Software Using Procedural Content Generation. In *Proceedings of the International Conference on Computer Safety, Reliability and Security (SAFECOMP)*, pages 33–44. Springer Berlin Heidelberg, 2013.

[ABG+05] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Mike Lowry, Corina Pasareanu, Grigore Roşu, Koushik Sen, Willem Visser, and Rich Washington. Combining Test Case Generation and Runtime Verification. *Theoretical Computer Science*, pages 209–234, 2005.

[ADY13] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-level Features for Robust Malware Detection in Android. In *Security and Privacy in Communication Networks (SecureComm)*, pages 86–103. Springer International Publishing, 2013.

[alc14] Kindsight Security Labs Report — H1 2014. `http://resources.alcatel-lucent.com/?cid=180437`, 2014. Alcatel-Lucent.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 259–269, 2014. ISBN 978-1-4503-2784-8.

[ASH+14] Daniel Arp, Michael Spreitzenbarth, Malte Huebner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[asm17] ASM: A Java Bytecode Engineering Library. `http://asm.ow2.org`, 2017.

[BBS+10] Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An Android Application Sandbox System for Suspicious Software Detection. In *Proceedings of the International Conference on Malicious and Unwanted Software (MALWARE)*, pages 55–62, 2010.

[BDS06] Marat Boshernitsan, Roongko Doong, and Alberto Savoia. From Daikon to Agitator: Lessons and Challenges in Building a Commercial Tool for Developer Testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 169–180, 2006. ISBN 1-59593-263-1.

[Bel05] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference*, pages 41–46, 2005.

[Bey16] Dirk Beyer. 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*, pages 887–904, 2016.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, 2002.

[BZNT11] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: Behavior-based Malware Detection System for Android. In *Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 15–26, 2011.

[CAM$^+$08] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 177–186, 2008.

[CBP$^+$11] Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: Model-inference-assisted Concolic Exploration for Protocol and Vulnerability Discovery. In *Proceedings of the USENIX Security Symposium*, 2011.

[CCY98] Tsong Y. Chen, Shing C. Cheung, and Siu Ming Yiu. Metamorphic Testing: A New Approach for Generating Next Test Cases. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, 1998.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

[CDPP96] David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. The Combinatorial Design Approach to Automatic Test Generation. *IEEE Software*, 13(5):83–88, 1996. ISSN 0740-7459.

[CGC12] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, volume 7459, pages 37–54. 2012. ISBN 978-3-642-33166-4.

[CHN12] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An Interpolating SMT Solver. In *Proceedings of the 19th International Workshop on Model Checking Software (SPIN)*, pages 248–254, 2012.

[CL11] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27:1–27:27, 2011.

[CR07] Feng Chen and Grigore Roşu. Mop: An Efficient and Generic Runtime Verification Framework. In *Proceedings of the International Conference on*

Object-oriented Programming Systems Languages and Applications (OOPSLA), pages 569–588, 2007.

[CSX08] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. *ACM Transactions on Software Engineering and Methodology*, pages 1–37, 2008. ISSN 1049-331X.

[DA14] Peter Dinges and Gul Agha. Solving Complex Path Conditions Through Heuristic Search on Induced Polytopes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 425–436. ACM, 2014. ISBN 978-1-4503-3056-5.

[DAUR15] Marko Dimjašević, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakamarić. Android Malware Detection Based on System Calls. Technical Report UUCS-15-003, University of Utah, School of Computing, 2015.

[DAUR16a] Marko Dimjašević, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakamarić. Evaluation of Android Malware Detection Based on System Calls. In *Proceedings of the International Workshop on Security and Privacy Analytics (IWSPA)*, pages 1–8, 2016. ISBN 978-1-4503-4077-9.

[DAUR16b] Marko Dimjašević, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakamarić. Evaluation of Android Malware Detection Based on System Calls — Dataset, 2016. URL `https://doi.org/10.5281/zenodo.154737`.

[DG15] Marko Dimjašević and Dimitra Giannakopoulou. Test-case Generation for Runtime Analysis and Vice Versa: Verification of Aircraft Separation Assurance. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 282–292, 2015. ISBN 978-1-4503-3620-8.

[DGH$^+$14] Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Zvonimir Rakamarić, and Vishwanath Raman. The Dart, the Psyco, and the Doop: Concolic Execution in Java PathFinder and its Applications. *SIGSOFT Software Engineering Notes*, 40(1):1–5, 2014. ISSN 0163-5948.

[DGR04] Nelly Delgado, Ann Quiroz Gates, and Steve Roach. A Taxonomy and Catalog of Runtime Software-fault Monitoring Tools. *IEEE Transactions on Software Engineering*, pages 859–872, 2004.

[Dim13] Marko Dimjašević. Automatic Testing of Software Libraries. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2013. Extended abstract.

[DLT13] Normann Decker, Martin Leucker, and Daniel Thoma. jUnit$^{\mathrm{RV}}$ — Adding Runtime Verification to jUnit. *NASA Formal Methods*, pages 459–464, 2013.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[DMSS12] Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. MADAM: A Multi-level Anomaly Detector for Android Malware. In *Proceedings of the International Conference on Mathematical Methods, Models and*

Architectures for Computer Network Security (MMM-ACNS), pages 240–253. 2012. ISBN 978-3-642-33704-8.

[DR13] Marko Dimjašević and Zvonimir Rakamarić. JPF-Doop: Combining Concolic and Random Testing for Java. In *Java Pathfinder Workshop*, 2013. Extended abstract.

[DRK+14] Morgan Deters, Andrew Reynolds, Tim King, Clark W. Barrett, and Cesare Tinelli. A Tour of CVC4: How it Works, and How to Use It. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design (FMCAD)*, page 7, 2014.

[EGC+10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[ELC10] Heinz Erzberger, Todd A. Lauderdale, and Yung-Cheng Chu. Automated Conflict Resolution, Arrival Management and Weather Avoidance for ATM. In *Proceedings of the 27th International Congress of the Aeronautical Sciences*, 2010.

[emu15] Emulab FAQ. `https://wiki.emulab.net/wiki/FAQ`, 2015.

[EOMC11] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the USENIX Security Symposium*, pages 315–330, 2011.

[FA11] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 13th European Software Engineering Conference held jointly with 19th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 416–419, 2011.

[FADA14] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based Detection of Android Malware Through Static Analysis. In *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 576–587, 2014. ISBN 978-1-4503-3056-5.

[FCH+11] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, pages 627–638, 2011.

[fdr15] F-Droid, Free and Open Source Android App Repository. `https://f-droid.org/`, 2015.

[FE07] Todd C. Farley and Heinz Erzberger. Fast-time Simulation Evaluation of a Conflict Resolution Algorithm Under High Air Traffic Demand. In *Proceedings of the 7th USA/Europe Air Traffic Management R&D Seminar*, 2007.

[FHE$^+$12] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*, pages 3:1–3:14, 2012. ISBN 978-1-4503-1532-6.

[FJC$^+$10] Matt Fredrikson, Somesh Jha, Mihai Christodorescu, Reiner Sailer, and Xifeng Yan. Synthesizing Near-optimal Malware Specifications from Suspicious Behaviors. In *Proceedings of the Symposium on Security and Privacy (SP)*, pages 45–60, 2010.

[GBS$^+$11] Dimitra Giannakopoulou, David H. Bushnell, Johann Schumann, Heinz Erzberger, and Karen Heere. Formal Testing for Separation Assurance. *Annals of Mathematics and Artificial Intelligence*, 63(1):5–30, 2011.

[GFA13] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Improving Search-based Test Suite Generation with Dynamic Symbolic Execution. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 360–369, 2013.

[GGJ$^+$10] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test Generation through Programming in UDITA. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 225–234, 2010.

[GGZ$^+$13] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing Non-adequate Test Suites Using Coverage Criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 302–313. ACM, New York, NY, USA, 2013. ISBN 978-1-4503-2159-4. URL `http://doi.acm.org/10.1145/2483760.2483769`.

[GHI$^+$14] Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Todd Lauderdale, Zvonimir Rakamarić, and Vishwanath Raman. Taming Test Inputs for Separation Assurance. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 373–384, 2014.

[GIB$^+$13] Pranav Garg, Franjo Ivančić, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. Feedback-directed Unit Test Generation for C/C++ Using Concolic Execution. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 132–141, 2013.

[GKC13] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT Solver for Nonlinear Theories over the Reals. In *Proceedings of the 23rd International Conference on Automated Deduction (CADE)*, pages 208–214, 2013.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223, 2005. ISBN 1-59593-056-6.

[GLM12] Patrice Godefroid, Michael Y. Levin, and David Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1):20:20–20:27, 2012. ISSN 1542-7730.

[GM15] Marco Gario and Andrea Micheli. pySMT: A Solver-agnostic Library for Fast Prototyping of SMT-based Algorithms. In *Proceedings of the 13th International Workshop on Satisfiability Modulo Theories (SMT)*, 2015.

[GTGZ14] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking App Behavior Against App Descriptions. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1025–1035, 2014. ISBN 978-1-4503-2756-5.

[GYAR13] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural Detection of Android Malware Using Embedded Call Graphs. In *Proceedings of the Workshop on Artificial Intelligence and Security (AISec)*, pages 45–54, 2013.

[GZZ$^+$12] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[Hav14] Klaus Havelund. Rule-based Runtime Verification Revisited. *International Journal on Software Tools for Technology Transfer*, pages 1–28, 2014.

[HGR13] Falk Howar, Dimitra Giannakopoulou, and Zvonimir Rakamarić. Hybrid Learning: Interface Generation through Static, Dynamic, and Symbolic Analysis. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 268–279, 2013.

[HHW$^+$13] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A Scalable System for Detecting Code Reuse among Android Applications. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 62–81. 2013. ISBN 978-3-642-37299-5.

[HR04] Klaus Havelund and Grigore Roşu. An Overview of the Runtime Verification Tool Java PathExplorer. *Formal Methods in System Design*, pages 189–215, 2004.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, 2nd edition, 2009.

[IX08] K. Inkumsah and Tao Xie. Improving Structural Testing of Object-oriented Programs via Integrating Evolutionary Testing and Symbolic Execution. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 297–306, 2008.

[jac17] JaCoCo Java Code Coverage Library. `http://www.jacoco.org/jacoco`, 2017.

[JAS14] Ranjith Kumar Jidigam, Thomas H. Austin, and Mark Stamp. Singular Value Decomposition and Metamorphic Detection. *Journal of Computer Virology and Hacking Techniques*, 11(4), 2014.

[JHG09] Karthick Jayaraman, David Harvison, and Vijay Ganesh. jFuzz: A Concolic Whitebox Fuzzer for Java. In *Proceedings of the 1st NASA Formal Methods Symposium (NFM)*, pages 121–125, 2009.

[JKXC10] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. OCAT: Object Capture-based Automated Testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 159–170, 2010.

[jun13] Third Annual Mobile Threats Report: March 2012 through March 2013. `http://www.juniper.net/us/en/local/pdf/additional-resources/3rd-jnpr-mobile-threats-report-exec-summary.pdf`, 2013. Juniper Networks Mobile Threat Center.

[JWH13] Gareth James, Daniela Witten, and Trevor Hastie. *An Introduction to Statistical Learning: With Applications in R*. Springer Texts in Statistics. Springer, 2013. ISBN 9781461471387.

[JZAH14] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. Morpheus: Automatically Generating Heuristics to Detect Android Emulators. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 216–225, 2014.

[KĐ07] James K. Kuchar and Ann C. Đrumm. The Traffic Alert and Collision Avoidance System. *Lincoln Laboratory Journal*, 16(2):277–296, 2007.

[KCK$^+$09] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. Effective and Efficient Malware Detection at the End Host. In *Proceedings of the USENIX Security Symposium*, pages 351–366, 2009.

[KJ13] Max Kuhn and Kjell Johnson. *Applied Predictive Modeling*. Springer, 2013. ISBN 9781461468493.

[KLS$^+$11] Kari Kähkönen, Tuomas Launiainen, Olli Saarikivi, Janne Kauttio, Keijo Heljanko, and Ilkka Niemelä. LCT: An Open Source Concolic Testing Tool for Java Programs. In *Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, pages 75–80, 2011.

[KPV03] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer-Verlag, 2003. ISBN 3-540-00898-5.

[LBK$^+$10] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. AccessMiner: Using System-centric Models for Malware Protection. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, pages 399–412, 2010. ISBN 978-1-4503-0245-6.

[LDG$^+$16] Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. JDart: A Dynamic Symbolic Analysis Framework. In *Proceedings of* *the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 442–459, 2016. ISBN 978-3-662-49674-9.

[LKM$^+$13] Shuying Liang, Andrew W. Keep, Matthew Might, Steven Lyde, Thomas Gilray, Petey Aldous, and David Van Horn. Sound and Precise Malware Analysis for Android via Pushdown Reachability and Entry-point Saturation. In *Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 21–32, 2013. ISBN 978-1-4503-2491-5.

[LS09] Martin Leucker and Christian Schallhart. A Brief Account of Runtime Verification. *The Journal of Logic and Algebraic Programming*, pages 293–303, 2009.

[LSM14] Shuying Liang, Weibin Sun, and Matthew Might. Fast Flow Analysis with Gödel Hashes. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 225–234, 2014.

[McM11] P. McMinn. Search-Based Software Testing: Past, Present and Future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163, 2011.

[mon17] UI/Application Exerciser Monkey. `http://developer.android.com/tools/help/monkey.html`, 2017.

[MT08] David McNally and David Thipphavong. Automated Separation Assurance in the Presence of Uncertainty. In *Proceedings of the International Congress of the Aeronautical Sciences*, 2008.

[OG10] Markus Ojala and Gemma C. Garriga. Permutation Tests for Studying Classifier Performance. *Journal of Machine Learning Research*, 11:1833–1863, 2010. ISSN 1532-4435.

[PBCK13] Sirinda Palahan, Domagoj Babić, Swarat Chaudhuri, and Daniel Kifer. Extraction of Statistically Significant Malware Behaviors. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 69–78, 2013.

[PHM$^+$09] Thomas Prevot, Jeffrey Homola, Joey Mercer, Matt Mainini, and Christopher Cabrall. Initial Evaluation of Air/Ground Operations with Ground-based Automated Separation Assurance. In *Proceedings of the 8th USA/Europe Air Traffic Management R&D Seminar*, 2009.

[PHP99] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-data Generation Using Genetic Algorithms. *Software Testing, Verification and Reliability (STVR)*, 9(4):263–282, 1999.

[PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed Random Test Generation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 75–84, 2007. ISBN 0-7695-2828-7.

[PMB+08] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 15–26, 2008. ISBN 978-1-60558-050-0.

[PMRB09] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A Fistful of Red-pills: How to Automatically Generate Procedures to Detect CPU Emulators. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[Pra16] Ignatius S. W. B. Prasetya. Budget-Aware Random Testing with T3: Benchmarking at the SBST2016 Testing Tool Contest. In *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*, pages 29–32, 2016. ISBN 978-1-5090-2205-2.

[PRV11] Corina S. Pasareanu, Neha Rungta, and Willem Visser. Symbolic Execution with Mixed Concrete-symbolic Solving. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 34–44, 2011.

[r17] The R Project for Statistical Computing. `http://www.r-project.org`, 2017.

[RFC13] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A System Call-centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors. *Proceedings of the European Workshop on System Security (EuroSec)*, 2013.

[ris14] RiskIQ's Report on Malicious Mobile Apps. `http://www.riskiq.com/company/press-releases/riskiq-reports-malicious-mobile-apps-google-play-have-spiked-nearly-400`, 2014.

[RJGV16] Urko Rueda, René Just, Juan P. Galeotti, and Tanja E. J. Vos. Unit Testing Tool Competition — Round Four. In *Proceedings of the International Workshop on Search-Based Software Testing (SBST)*, pages 19–28, 2016. ISBN 978-1-5090-2205-2.

[RTWH11] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic Analysis of Malware Behavior Using Machine Learning. *Journal of Computer Security*, 19(4):639–668, 2011.

[RWS+15] Robert Ricci, Gary Wong, Leigh Stoller, Kirk Webb, Jonathon Duerig, Keith Downie, and Mike Hibler. Apt: A Platform for Repeatable Research in Computer Science. *SIGOPS Operating Systems Review (OSR)*, 49(1):100–107, 2015. ISSN 0163-5980.

[SA06] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-checking Tools. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 419–423, 2006. ISBN 978-3-540-37406-0.

[SB14] Nastaran Shafiei and Franck van Breugel. Automatic Handling of Native Methods in Java PathFinder. In *Proceedings of the International SPIN Symposium on Model Checking of Software*, pages 97–100, 2014. ISBN 978-1-4503-2452-6.

[SBdP11] Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S. Păsăreanu. CORAL: Solving Complex Constraints for Symbolic Pathfinder. In *Proceedings of the NASA Formal Methods Symposium (NFM)*, pages 359–374, 2011. ISBN 978-3-642-20397-8.

[SDTC+16] Tanuvir Singh, Fabio Di Troia, Visaggio Aaron Corrado, Thomas H. Austin, and Mark Stamp. Support Vector Machines and Malware Detection. *Journal of Computer Virology and Hacking Techniques*, 12(4):203–212, 2016. ISSN 2263-8733.

[sf113] The SF110 Benchmark Suite. `http://www.evosuite.org/experimental-data/sf110`, 2013.

[SGS+16] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, 2005.

[smt17] The SMT-LIB Standard. `http://smtlib.cs.uiowa.edu`, 2017.

[soo17] Soot: A Java Optimization Framework. `http://sable.github.io/soot`, 2017.

[SPG16] Abdelilah Sakti, Gilles Pesant, and Yann-Gaël Guéhéneuc. JTExpert at the Fourth Unit Testing Tool Competition. In *Proceedings of the International Workshop on Search-Based Software Testing (SBST)*, pages 37–40, 2016. ISBN 978-1-5090-2205-2.

[TH08] Nikolai Tillmann and Jonathan de Halleux. Pex—White Box Test Generation for .NET. In *Proceedings of the International Conference on Tests and Proofs (TAP)*, pages 134–153, 2008. ISBN 978-3-540-79123-2.

[Thi08] David Thipphavong. Analysis of Climb Trajectory Modeling for Separation Assurance Automation. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, 2008.

[TLN+14] Hien Thi Thu Truong, Eemil Lagerspetz, Petteri Nurmi, Adam J. Oliner, Sasu Tarkoma, N. Asokan, and Sourav Bhattacharya. The Company You Keep: Mobile Malware Infection Rates and Inexpensive Risk Indicators. In *Proceedings of the International Conference on World Wide Web (WWW)*, pages 39–50, 2014. ISBN 978-1-4503-2744-2.

[TXT+11] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. Synthesizing Method Sequences for High-coverage Testing. In *Proceedings of the International Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA)*, pages 189–206, 2011. ISBN 978-1-4503-0940-0.

[TZHS15] Haruto Tanno, Xiaojing Zhang, Takashi Hoshino, and Koushik Sen. TesMa and CATG: Automated Test Generation Tools for Models of Enterprise Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 717–720, 2015.

[vEG14] Christian von Essen and Dimitra Giannakopoulou. Analyzing the Next Generation Airborne Collision Avoidance System. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 620–635, 2014.

[WLS$^+$02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 255–270, 2002.

[WROR14] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, pages 1329–1341, 2014. ISBN 978-1-4503-2957-6.

[YXA$^+$15] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 303–313, 2015.

[YY12] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the USENIX Security Symposium*, pages 569–584, 2012.

[ZHT$^+$04] Zhi Quan Zhou, D. H. Huang, T. H. Tse, Zongyuan Yang, Haitao Huang, and T. Y. Chen. Metamorphic Testing and its Applications. In *Proceedings of the International Symposium on Future Software Technology (ISFST)*, pages 346–351, 2004.

[ZJ12] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the Symposium on Security and Privacy (SP)*, pages 95–109, 2012.

[ZJS$^+$11] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. TaintEraser: Protecting Sensitive Data Leaks Using Application-level Taint Tracking. *SIGOPS Operating Systems Review (OSR)*, 45(1):142–154, 2011. ISSN 0163-5980.

[ZWZJ12] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.