Using design diversity and optimal adjudication for detecting malicious web scraping and malware samples

Pedro Daniel Magalhães Marques

Submitted for the degree of Doctor of Philosophy

City, University of London
Department of Computer Science

February 24, 2022

Abstract

Due to the constantly evolving nature of cyber threats and attacks, organisations see an ever-growing requirement to develop more sophisticated defence systems to protect their networks and information. In an arms race such as this, employing as many techniques as possible is crucial for companies to stay ahead of would-be attackers.

Design diversity is a technique with a significant history behind it, which has become more widely used as the availability of off-the-shelf defence software has become more commonplace. The simple concept behind design diversity is the age-old saying that ”two minds think better than one”. When combining multiple tools for cyber defence, it’s reasonable to expect that when these tools use different techniques, or work under different assumptions and configurations, they would also likely detect different threats. Hence, the security events that one tool misses or misclassifies, the other could correctly handle, and vice-versa. We would expect design diversity to remain an important design paradigm for as long as building a completely foolproof security system stays within the realms of impossibility.

While design diversity is appealing, and it has been used to great success in the past, it is important to realise that any possible gains from using this technique are entirely dependent on how diverse the various tools are, and on the context in which it is applied. Applying design diversity will yield different results in different environments, so it is important that empirical results are provided in as many contexts as possible.

In this work, we have looked at the use of design diversity in two major contexts. The first context deals with the question of detecting malicious web scraping activity. We have analysed three separate datasets provided to us by Amadeus - a global provider for the travel and tourism industry - which contain the HTTP traffic they observed within their network, as well as the alerts raised by two of their web scraping detectors. We studied how the combined performance potential of the two tools compares to their individual performances, in 1-out-of-2 (1oo2) and 2-out-of-2 (2oo2) adjudication schemes, meaning that a combined system raises an alert if any one of the internal tools does so as well, or the combined system only raises an alert if both internal tools do so, respectively. We’ve also identified several aspects that highlight the different alert patterns of both tools, which we use to explain the inherent diversity between the two.

The second context in which we have studied the use of design diversity is in the use of machine learning models for the classification of malware and benign software samples. We’ve done this with the use of a dataset that looked at the performance of 37 different RNN machine learning models used to classify a pool of over 4000 software samples, which originated from a previously published paper whose authors we have collaborated with. With the higher number and degree of diversity of the detection tools (the machine learning models) in this study, we were able to expand our results with additional adjudication schemes, anywhere between 1oo10 and 10oo10, as well as more interesting schemes, such as simple majority schemes, e.g., 3oo5. Similarly to the first body of work, we studied and summarised the different aspects that led to diversity in the behaviour of machine learning models.

When utilising multiple diverse systems, each producing a result, a voting or adjudication system is needed to decide on the overall system output/decision. The use of conventional adjudications schemes (e.g., 1-out-of-2) provides a useful first point for the use of design diversity, but these schemes may be deficient in comparison with others such as those that use *optimal adjudication.* As opposed to conventional adjudication schemes, where the individual outputs of each internal tool are not taken into account - i.e., a 1oo2 scheme does not care which one of its two internal tools raised an alert, only that one of them did - this is not the case for optimal adjudication. In optimal adjudication, the combined outputs of all the internal tools are called syndromes, and the output of the overall system is going to be dependent on which unique syndrome was generated for any given classification sample. This affords us several benefits, which we will detail in depth, primarily that specific tools can be given higher confidence over others, and that the error cost of generating false positive or false negative outputs can be taken into account when deciding the output of the overall system, such that we can optimise for the lowest error cost overall.

We have looked at the use of optimal adjudication in particular with our second dataset concerning the use of machine learning models for the classification of software samples. We expand on the benefits afforded over using conventional adjudication schemes, and delve into the aspects that make the various machine learning models diverse from one another.

We expect the results from this thesis will provide insight into the use of different adjudication schemes in the contexts we highlight (contexts in which, to the best of our knowledge, previous research has not been published), as well as provide guidance on the creation of such combined systems for use in security deployments beyond the contexts we have studied.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to extend my deepest gratitude to my supervisor, Dr. Ilir Gashi of City, University of London, for all the help and guidance he has provided to me over the course of this work. His direction made this thesis what it is, and I would not have been able to achieve what I have under the guise of anyone else.

I am extremely grateful to Prof. Alysson Bessani from the University of Lisbon, who acted as my supervisor during my MSc. thesis, and first introduced me to the staff at City and enabled me to come study in the United Kingdom, and subsequently endeavour in this PhD course.

A thank you to Zayani Dabbabi, Miruna-Mihaela Mironescu and Olivier Thonnard from Amadeus, for providing datasets used in this work, as well as their presence in frequent meetings used to discuss and gain insight into the analysis.

Similarly, a thank you to Matilda Rhode from the University of Cardiff and Airbus, for providing a dataset used in this work and offering plenty of assistance in the interpretation of the data and the direction of the analysis.

I would also like to express my gratitude to Dr. Kizito Salako of City, University of London, for his collaboration on the later portions of this thesis. He is responsible for enabling me to adequately carry out my work in the context of optimal adjudication, and he always made himself available whenever I needed assistance.

Thank you to my father, Américo Marques, for always encouraging and enabling me to pursue my education, and for the many big sacrifices he has made for mine and my brother's sake.

Finally, a big thank you to my girlfriend and soulmate, Ugnė Paukštytė. Her support throughout these past three years has been invaluable in allowing me to reach this finish line. Thank you for keeping me sane during the periods where I felt lost the most.

1 Introduction

1.1 Context of the research

All systems need to be reliable and secure. An unreliable system is one that is prone to failures, caused by random or deliberate uncovering of faults and vulnerabilities, and the effects of these failures can range from mere annoyance, in the case of day-to-day systems such as word processors, to outright catastrophic consequences when dealing with safety-critical scenarios, such as flight control. Increasing the reliability and security of a system is a critical field of study in computer science.

The topic of reliability, and the techniques employed to increase the reliability of software systems has been extensively studied over the years [4]. The use of replication and redundancy techniques are commonly used to increase reliability, especially in hardware. These techniques yield the best results in hardware because hardware components are subjected to physical decay - such as rust or moisture - and as such, component failures are independent from one another. When applied to software however, replication and redundancy offer limited benefits, because the conditions that lead to the failure of one component will also lead to the failure of another identical component. Because of this, software was historically made more reliable through fault prevention and reduction efforts. While these techniques have merit, and are necessary, on their own they may not be sufficient to deliver high reliability and security. To achieve higher levels of reliability and security, software fault tolerance, through the use of diverse redundancy, has been prescribed in the past [1, 5, 6].

A popular technique which has a storied past is that of *N-version programming.* N-version programming (NVP) is a process through which multiple variants of the same system are constructed independently, and then joined into a combined system as a complete product. This final product is thus a composition of multiple, diverse systems which perform the same function, but do so through different means and algorithms. The core idea in this pursuit is that the different variants will fail on different demands, and where one fails, another might not. N-version programming is widely used in highly-critical scenarios where the consequences of failure are disastrous, such as flight control systems and rail traffic [7, 8], and it has achieved significant positive effects. One of the obvious drawbacks to the use of NVP techniques is that it carries an additional, potentially high cost of development. While there are studies suggesting that the cost of developing NVP variants is not as severe as one might imagine at first [9], these are far and few between. Hence more studies are needed to justify the benefits, in increased reliability and security, on the use of NVP for a particular domain.

The growing abundance and ubiquity of off-the-shelf (OTS) software components built for myriads of applications has created a secondary alternative to that of NVP, known as *design diversity.* These OTS components are cheaper to acquire (possibly even free as open-source software becomes increasingly normalised), and already provide out of the box, tried and tested methods of operation. A developer looking to put together a system that uses design diversity by utilising multiple OTS components needs only to build the glue code (*middleware*) which makes sure the different products work together to achieve the desired goal. This means that a system can be constructed at a fraction of the cost, while still providing similar reliability benefits to those found in N-version programming.

In both N-version programming and design diversity, the potential benefits to reliability are justifiable through common sense: the advantages and drawbacks of each individual component can be leveraged such that the failures of one are covered by the correct operation of another. However, the degree to which the reliability benefits are observed is largely dependent on two factors: 1) the actual diversity between the components, and whether they tend to fail dependently or independently from one another and; 2) the voting scheme being utilised to adjudicate between the outputs of the various components.

An adjudication scheme determines how the overall system operates, based on how its internal components perform. In the simplest example, in which a system is composed of two individual components, each generating outputs if certain conditions are met, there are two basic adjudication schemes that can be chosen. The first is a 1-out-of-2 scheme (1oo2), where the overall system generates outputs if either of the internal components does so. Under this scheme, if one of the internal components fails to correctly generate an output under a certain set of conditions, the second component can still be used to overcome that error. However, it also means that if a component incorrectly generates an output when it shouldn't, that error will be propagated to the overall system, thus generating a "false positive". The other possible scheme is a 2-out-of-2 (2oo2), where the overall system only generates an output if both of its internal components agree that such an output should be generated. This has an opposite effect on the reliability measures of the system. It decreases the number of "false positives", but at the same time means that outputs that should have been generated were missed because of a single failed component. The choice of which adjudication scheme to utilise is one whose answer is highly dependent on the context in which it is applied, and which types of failures (false positives or false negatives) lead to greater losses.

The question of reliability, and the techniques we have discussed also apply when it comes to the *security* of a system. Cybercrime and cyber-attacks are a growing concern to organisations and institutions worldwide. Over the years, the potential monetary value that can be gained from carrying out attacks has increased substantially, driven up by the increased reliance organisations have on their technology and information systems [10]. For the security of a system, the benefits of N-version programming and design diversity are also dependent on the types of attacks the system is exposed to [5]. Because of this, empirical results for each field of study are a necessity for organisations to be able to make informed, empirically supported, decisions on how best to increase the security of their systems.

The first section of our work was done in the context of detecting malicious web scraping activity, in collaboration with Amadeus. They have provided us with three datasets collected from their internal networks in different locations, which included the alerts generated by two of their intrusion detection systems tailored to monitor for suspicious web scraping activity. We have looked at the use of conventional adjudication schemes with these datasets and highlight the diversity that existed between the two security tools. This collaborative work was done under the auspices of the DiSIEM project, which was a project that aimed to enhance the capabilities of SIEM (Security Information and Event Management) systems by improving the quality of security events collected from diverse sensors [11].

Our next section of work deals with the context of using machine learning algorithms for the classification of benign and malicious software samples. We have analysed a dataset of 37 different RNN machine learning models used to classify a pool of over 4000 software samples. Due to the higher number of classification models in this dataset we were able to expand our research to included higher numbers of adjudication schemes. This second section of work was done in collaboration with researchers at the University of Cardiff, as a follow up of a previous research they had published on the use of these same RNN models [12].

Finally, we expand on both contexts with the use of optimal adjudication techniques. Optimal adjudication differs from conventional adjudication schemes (e.g., 1oo2 or 2oo2), by taking into account the outputs of each individual internal classifier, rather than simply a quorum of them. By doing this, we are able to place higher levels of confidence on individual classifiers, as well as adapt our adjudication outputs based on varying error costs from generating false positive or false negative classifications. We detail our methodology for constructing optimal adjudicators, as well as the improvements observed over conventional adjudication schemes.

1.2 Objectives

The objectives of this research are as follows:

- 1) Assess the effects, and present the results on the use of design diversity on the security of systems. This was done in:
  - 1.1) the context of malicious web scraping detection, in collaboration with Amadeus, an industrial partner which provides support and infrastructure for the global travel and tourism industry and provided us with real datasets of network traffic observed within their networks and the alerts generated by two of their internal intrusion detection tools.
  - 1.2) the context of malware sample detection using machine learning methods, in collaboration with Matilda Rhode from the University of Cardiff, who provided us with a dataset of the classifications generated by 37 different recurrent neural network models when classifying a pool of over 4000 malware and benign software samples.
- 2) Assess and compare the benefits and drawbacks of different adjudication/voting schemes when applied to systems using design diversity, and the effects that different configurations can have on the overall reliability observed. In particular, the use of optimal adjudication schemes versus non-optimal adjudication schemes were assessed for both contexts we've highlighted above.
- 3) Describe a clear methodology that can be followed for analysing the effects that diverse tools can achieve when paired together.

1.3 Contributions of the research

This research has provided valuable information to our industrial partner, Amadeus, on the effects observed due to the use of design diversity in their system. This includes the general improvements found in their systems due to the use of design diversity, as well as insight into how further improvements can be achieved. Along with this, a visualisation dashboard has been provided to Amadeus, which is used by their SOC teams in order to inform their decisions about the security of their system.

Our work has also contributed to the DiSIEM project, through direct discussion and sharing of results with members of the project. DiSIEM is a project funded by the European Union with the goal of enhancing SIEM systems by improving the quality of the events collected by diverse sensors and adding support for OSINT data sources.

From a more general perspective, our research provides empirical evidence on the benefits of design diversity for the detection of malicious web scraping activity, as well as the use of machine learning for the classification of software samples, both contexts which, to the best of our knowledge, have not received in-depth research. We demonstrate how these benefits can be improved with the use of optimal adjudication schemes compared to conventional schemes. Additionally, we present our methodology for achieving this results, which can be used by other researchers to analyse the effects of design diversity in other contexts.

During the first year of the research, a regular research paper was accepted and presented at the 23rd IEEE Pacific Rim International Symposium on Dependable Computing, in Taiwan, in December of 2018. [13]. This paper presents the analysis of the first dataset provided to us by Amadeus and reports our findings on the use of design diversity for tackling malicious web scraping. An earlier Fast Abstract that explained some of the work was presented at the IEEE International Conference on Dependable Systems and Networks conference, in 2018 [14].

Our second section of work, dealing with the use of machine learning for the classification of software samples has resulted in the creation of an additional journal paper, detailing the results obtained with the use of conventional adjudication schemes. This paper has been published in the journal of Computers and Security [15]. Additionally, further work looking at the application of optimal adjudication in this context is currently underway in collaboration with Dr. Kizito Salako from City, University of London, and a journal submission is expected by mid-2022.

1.4 Thesis outline

The remainder of this thesis is outlined as follows: in Section 2 we present a review of the current literature, looking at system reliability and the techniques used to improve it, fault tolerance, as well as the issues of general malware, malware detection, cyber-attacks and web scraping; in Section 3 we review the methodology we have employed in our research efforts; Section 4 focuses on our collaboration with Amadeus, and our study of the use of design diversity in the context of detecting malicious web scraping; section 5 expands on work done by Matilda Rhode from the University of Cardiff, and relates to the use of design diversity with machine learning used to classify software samples as either malicious or benign; the research presented in these two previous sections is then expanded in section 6 by looking at the benefits of optimal adjudication and its effects in both of these contexts; finally, in section 7 we conclude this thesis with discussion of our three major research endeavours, a review of our research objectives and detail some of the work that could be further pursued.

2 Literature review

In this section we discuss relevant literature for this research, with a focus on software reliability and security, and techniques to enhance this, namely, N-version programming and design diversity. We discuss the different adjudication schemes possible when setting up such systems, and various metrics used for assessing the performance of classifiers. Further, we delve into the realms of malware and malware detection, including web threats and malicious web scraping.

Our literature review process informally follows the Systematic Literature Review (SLR) process proposed by Barbara Kitchenham [16]. We collected research from prominent sources, both journals and conferences, including: International Symposium on Software Reliability Engineering (ISSRE); Transactions on Dependable and Secure Computing (TDSC); European Dependable Computing Conference (EDCC); International Conference on Computer Safety, Reliability and Security (SAFECOMP); International Conference on Dependable Systems and Networks (DSN); and from these identified primary studies. We have also looked at research performed by members at City, University of London, and included those in our efforts. When looking for literature we have focused on keywords most related to our own research (design diversity, bot scraping detection, security assessment, malware detection, etc). For some topics - namely system reliability and security, N-version programming and design diversity - we have tried to review as far back as we could find research on, in order to paint a historical picture of the field of study. For topics which originated outside the field of computer science, we have also attempted to find relevant information. These include binary and non-binary classification systems, topics which appear frequently in the medical and machine learning fields.

For studies which presented empirical data, we attempted to collect a breath of different research areas. For example, for studies on the effects of design diversity, we have attempted to identify the broadest number of areas for which empirical results exists (i.e., use of design diversity in AntiVirus solutions, IDSs, firewalls, etc.). What we confirmed when looking at these empirical studies, which we discussed during our literature review chapter, is that the effects of design diversity can vary significantly from one field to another, and thus, comparing the results of different studies between themselves does not necessarily yield useful information. For this reason, we have not followed the data extraction and synthesis steps of Kitchenham's SLR process.

During our literature collection we identified four other SLRs which were of interest to our own research. These were: i) a systematic literature review on the design of different adjudication systems for fault-tolerant Service Oriented Architectures (SOAs) by Nascimento et al. [17]; ii) a survey of different approaches for detecting bot traffic made against web servers, by Acarali et al. [18]; iii) and iv) two separate surveys on malware detection techniques, by Vinod et. al. [19] and Saeed et. al. [20].

2.1 Software reliability and security

The reliability of a system can be defined as the ability of that system to continue providing a correct service [21] - "correct" in this instance means that its results conformed to the expected behaviour as specified in a system specification. All software systems require a certain level of reliability. Simple applications, such as word processors need a high enough reliability such that it allows users to work with them at a reasonable pace, without being affected by constant crashes or mishaps. On the other end of the scale, more critical systems, such as air flight control software require very high levels of reliability, as faults in their design or operation have the potential to lead to catastrophic outcomes.

There are two main issues when dealing with system reliability. The first of these is how one goes about achieving the required level of reliability for any one system. What techniques should be applied, and how can one do this effectively and efficiently? The second issue is how to assess the level of reliability a system has actually achieved, such that one can be convinced that a piece of software will meet its reliability target in a given production environment.

The reliability of a system goes down as the system's failure rate increases. It is important therefore, that we describe the process that leads to a system failure. In Fig. 1, we illustrate the commonly accepted model of how a failure arises in a software system [21,22].

$$\begin{tikzcd}\boxed{\textbf{Fault}} \arrow[r] & \boxed{\textbf{Error}} \arrow[r] & \boxed{\textbf{Failure}}\end{tikzcd}$$

Figure 1: Fault, error, failure model

The first step in the chain is a fault. A fault is an avoidable defect which is left in the system due to a human mistake[1]. If triggered, this fault causes the particular algorithm in which it is present to produce an undesired result, which leads to an erroneous state - an error. While the system is in this erroneous state, it does not necessarily lead to an immediate, externally observable failure, it only means that it has the potential to do so. It is when the system interacts with the erroneous state - a variable which contains an incorrect value, for example - that it leads to a failure of the system, either fatal or not. Additionally, this model can be chained, if we classify a system as a set of different components. This means that a component which contains a fault can produce an error, leading to its failure, and in turn this failure may trigger a fault in another component.

The goal with increasing system reliability is to minimise the number of failures observed in a system. There are two major types of failures that we see in systems. The first, and more straightforward of these are physical failures. These failures arise in hardware because of physical decay of the components over time - e.g. rust, burnt elements, build-up of moisture, etc. These failures have historically been deemed unavoidable, because there will always exist physical decay, as components are exposed to the elements and the laws of physics. As such, a large part of the reliability achieved in the physical systems and components comes from the use of redundancy - the use of multiple, identical components.

The second major type of failures are known as "systematic" failures, as defined by Strigini et. al [6], and pertain to software, and the logical components of a system. Systematic failures occur, as the name implies, systematically whenever a specific set of circumstances occurs in the use of products. These failures arise, not from physical decay, but from human faults, left in a system during its design or development phase, either

[1]A fault may also be *intentionally* planted in the system, with or without maliciousness. We will touch on this later on.

because of oversight, or lack of foresight when building these. The nature of systematic failures could lead one to expect that they are deterministic, and as such the use of probabilistic models would not yield significant results. However, while the failure process is indeed deterministic - a system always fails under a certain set of circumstances and parameters - the occurrence and triggering of faults are not. We can take advantage of this, to build a probabilistic model of fault distribution present in a system.

Littlewood et. al describe this model of thinking [1], for two distinct types of systems. For real-time systems, which are constantly running, we can define a time variable (not necessarily a real time clock) and embed arising software failures on this timeline. What we hope to accomplish is to obtain an assurance that the rate of failure is sufficiently low, or that there is a high probability of a system surviving a pre-defined operation period without failures arising.

For safety systems, such as a nuclear reactor protection system, which only respond on occasional demands from a larger environment, we can model their reliability as the number of failed demands over the sequence of all demands made to the system. This is usually expressed as the probability of failure upon demand (pfd).

Figure 2 gives us a high level overview of this model. On the left side we have a set of all demands, D, which can be processed by a particular system. Each point in this set represents a vector of the possible values which could affect the application process - for example, in the context of a nuclear reactor protection system, each point could be a vector of temperature values, pressure values, current time, etc. A program, P, translates all of these points in the demand space, into points in the output space. There are two divisions that we make, on both the demand space, and the output space. The straightforward objective is that each point in the demand space translate into an acceptable output after being processed by the program. However, we need to model the failures observed by the system, and as such, there is a subset of demands (Df) from the demand space, which translate into unacceptable outputs.

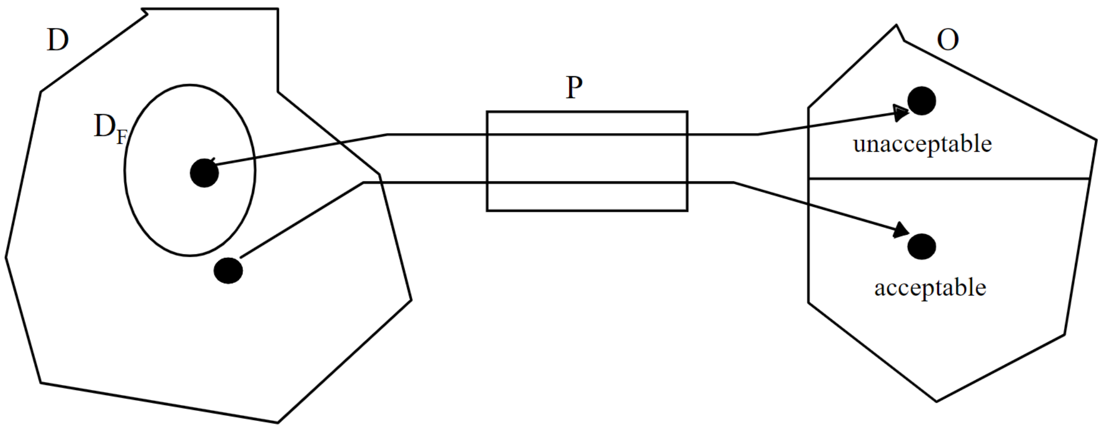


Figure 2: Software failure model [1]

There are some significant problems when applying this model in practice. The first of which is that, modelling the entirety of the demand space of a program is exceedingly difficult for any non-trivial system. Any real-world application for which the use of software reliability techniques would have a significant impact would have far too large of a demand space to describe theoretically, let alone meaningfully test. The second issue is that, while we have modelled the demands which translate to unacceptable outputs as a single cluster, making this clear division in a real-world system is an obvious challenge to undertake, and there are no guarantees that the various "problematic" demands will even have anything in common between themselves. Identifying these demands would be, in a real-world scenario, a mostly case-by-case endeavour.

The reliability models and concepts we have described so far can be extended to encapsulate concepts of security, and thus increase system reliability when presented with attacks. The MAFTIA project defines security as the concurrent existence of three system properties: availability - readiness for a correct service; confidentiality - prevention of unauthorised access to information; and integrity - prevention of unauthorised state alteration. [21]. If any of these properties of a system are compromised, so is the reliability of the system, as it cannot provide the correct and intended service.

A useful conceptual extension to the fault-error-failure for security is presented in [21]. In Fig. 3 we add two new stages, attack and vulnerability. The process that leads to a failure now starts with an attack by a malicious agent. This attack targets a specific vulnerability present in the system, one which allows access for the attacker to cause a fault - for example, through code injection. The path then follows its natural process, where a fault causes an error in the system, which subsequently leads to a failure.

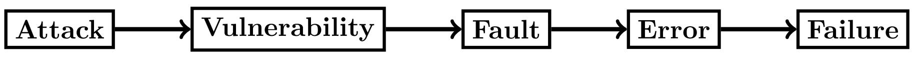


Figure 3: Fault, error, failure model extended for security

One of the biggest differences here is that of "intentionality". We are no longer dealing with faults caused through random operation, but with an agent who is purposefully attempting to cause a system failure. Previously, when dealing with just reliability and not security, we would usually work in terms of *failures over time.* However, for security, this notion of time loses most of its meaning, since clearly, a system might be able to operate correctly indefinitely, so long as there are no attacks made against it. Littlewood et. al argue that the use of an effort function, rather than a time one, is more appropriate in this aspect [23]. This is for several reasons:

- Firstly, for the case of intentional breaches arising from intentional malicious faults, the notion of time is meaningless, because the effort is expended in originally inserting the fault, which occurs before the operational system exists, and exploiting it afterwards is likely trivial for an attacker.

- Secondly, there are cases where a time variable is absent and where the expenditure of effort is instantaneous, such as the use of bribery. The chance of success for the bribery would clearly depend upon the magnitude of the bribe (effort), which is difficult to translate into time.

- Finally, the notion of effort can later be translated into a time function, should one be needed, for example, to assert that there will be no security breaches during the lifecycle of the system.

The occurrence of an attack is dependent on two main points. Firstly, there is the reward that an attacker might expect to acquire by carrying out a successful attack. This is fairly easy to assess, as one can usually assign some sort of monetary value of a successful breach (e.g., the value of company information). The second point, and the more difficult to calculate is the effort required by an attacker to carry out a successful attack. This effort is dependent on multiple factors: the experience and education of the attacker, the amount of money or time available to the attacker, the quality of the system in question, additional security mechanisms utilised, etc. One of the bigger variables to measuring the effort required stems from the vulnerabilities present in the system.

There are three main types of vulnerabilities that a system can possess. Firstly, there are accidental vulnerabilities. These are simply errors made during the design or development of a system. Then there are intentional vulnerabilities, which can be divided into two types. Malicious intentional vulnerabilities are vulnerabilities placed within a system, with the intention of carrying out an attack later on. These include *trojan horses* or *backdoors*. Then there are non-malicious intentional vulnerabilities. These are created, sometimes without knowledge, through certain design decisions. For example, a system developer might decide that a certain security feature is too costly for the system to operate with and choose to trade-off security for performance.

The concept of security and malicious attacks also has an influence on the software failure model presented in Fig. 2. Previously, the *demand profile* could be drawn too tightly, taking into account only the system. When dealing with security however, one should draw the demand profile past the system itself, and involving other systems or agents, and in this case, the potential attackers. This means that the demand profile of a system is increased, primarily due to attackers being able to think of "demands" developers thought impossible. Furthermore, the demand profile of a system is also constantly changing, as attackers learn the system, exhausting certain avenues of demands which do not lead to successful breaches, and focusing on the more likely branches to attack.

2.1.1 N-version programming

The concept of N-version programming was first introduced by Avižienis in 1977 [24]. It describes the process where multiple pieces of independent, functionally equivalent software are created from the same initial specification and are then used in parallel to assess the same input and their output adjudicated on by a consensus process. The result is a system which uses diverse components and is intended to achieve a higher reliability.

The use of N-version programming has seen growing adoption since its proposal in 1977, and this is especially true for safety-critical systems, with examples such as railway interlocking and train control [8], Airbus flight controls [7] and the protection of the Darlington nuclear reactor [25]. Multiple studies have also been carried out to test the viability and benefits of the use of n-version programming and design diversity [26–28].

The main idea behind the use of multiple pieces of software stems from the assumption that different versions will fail on different inputs. While there is some contention as to whether this assumption is a valid one, it is generally accepted that software developed independently does not adhere to the models that assume failure independence. The general reasoning is that some problems are inherently more difficult than others, and thus different versions will more likely fail coincidentally when tackling these.

A famous experiment on the correctness of the assumption that independent versions fail independently was carried out by Knight and Leveson in 1986 [29]. The experiment involved the creation of 27 different versions of a missile launch interceptor program developed from the same specification document, which was carried out by students at different academic levels. The authors concluded that the assumption of independent failure did not hold true, with the versions created having a number of common failures between them which was higher than statistically expected if independence was assumed. The authors gave two major hypotheses as to why multiple versions shared the same failures:

- The first is that certain parts of any problem are simply more difficult than others, and thus programmers will tend to make similar mistakes when tackling them.
- The second is that the common failures may reflect flaws in the specification documents that were used to develop the versions from.

The authors point out however, that this does not mean that n-version programming should not be used, just that the additional reliability gained from using the technique may not be as high as the theoretically expected under the assumption of failure independence.

Another experiment on the assumption of independence was carried out in 1991, by Eckhardt et. al. [30]. In it, 20 teams made up of 2 members each, independently developed an aerospace application, complex in nature, with each team having at least one expert in both computer science and mathematics. The aim of the study was to compare the results of the probability of failure of sets of versions against the theoretical models of independence failure proposed by the same authors previously [31]. The authors concluded that there were modest gains in reliability using redundant software in N-version architectures, and that N-version systems are effective at coping with failures of a few bad programs. However, under operating conditions where individual programs were more reliable, the failures that did occur tended to be coincident, and this "greatly exceeded the rates expected by chance under the assumption of independence". The results are comparable to those found in the Knight and Leveson experiment.

In the 80s and 90s there was considerable debate in the research community about the Knight and Leveson results and what those results meant for the adoption and assessment of n-version programming. Avižienis [28] criticised the Knight and Leveson experiment due to the (lack of sufficient) effort from the programmers, the background of the programmers involved (all programmers were students - although some with extensive programming experience), and the lack of complexity of the programs developed (average size of the versions developed was just over 500 lines of code). Hence the criticism was that the Knight and Leveson experiment was not representative of real-world, safety-critical applications. However, Avižienis' own experiment in 1988 did not provide a significant increase in program complexity, with their teams averaging under 1000 lines of code when disregarding empty lines and comments. Additionally, as pointed out by Knight and Leveson in their reply to criticism [32], Avižienis' experiment focuses on coincident faults rather than coincident failures. The important distinction here is that noncoincident faults in programs can still lead to coincident failures during operational time, which in turn means that independent faults may not necessarily represent an increase in reliability.

There are some areas of N-version programming to which the literature agrees on. Firstly, most authors agree that the use of multiple, independent programming teams is a base requirement for achieving an acceptable level of diversity between versions. Additionally, other forms of imposed diversity can easily lead to additional diversity in the resulting programs, such as requiring different programming languages, algorithms, etc.

Secondly, and more importantly, it is commonly agreed that the use of the same specification document is a major cause of coincident failures in multiple versions. Not only due to the fact that specification faults will propagate to all developing teams, but also in the way that a specification can easily lead to less variety in the way programmers reconcile with the problem at hand, and how they approach developing the application. For example, as pointed out in [28], "even when coding independent computations that could be performed in any order [...] algorithms specified by figures were generally implemented by following the corresponding figure from top to bottom". Avižienis advocates that specification documents should be reduced as much as possible to the "what" rather than "how". He also points out that independence of faults is an **objective** and **not** an **assumption** of the N-version programming approach [33].

Concerns with the use of N-version programming when adjudicating between multiple correct results (MCR) have been expressed before [34], especially when it concerns the comparison of finite-precision calculations [35]. Some studies done on the use of N-version programming may have been hiding potential reliability and security benefits due to these cases where MCR or finite-precision are required. For example, consider the case where programs are expected to output the altitude of an aircraft (such as the case in the Eckhardt et al. experiment mentioned previously), where two versions output the correct value while 3 others output unique, incorrect values. In this case one could argue that having two replicas agree on the same value is a better indication of correctness of output. This error detection was not taken into account during the Eckhard et. al. experiment and is one that could lead to an increase in reliability. Some of these issues can be dealt with by using different adjudication schemes (which we discuss in a future section), however, they are not present when N-version programming or design diversity are applied to binary problems.

2.1.2 Design diversity and defence in depth

Another more general term for n-version programming is *design diversity*. In parts of the literature the two terms are used interchangeably: techniques that make use of diverse components and pieces of software to increase the overall reliability of systems. In this thesis we will make use of the term design diversity to mean the use of different products that were not necessarily built with the intent of working alongside each other, and thus are not sourced from the same specification, but that largely perform the same function. Another term, used widely in the security literature is *defence-in-depth*. The meaning is similar to design diversity - the use of multiple defence systems to reduce the probability of failure (e.g., successful attack) - through the use of functionally diverse systems to reduce this probability (for example, a Firewall, an IDS and an antivirus). Another term used in the literature is defence-in-breadth, which is closer to the traditional notion of design diversity (e.g., use of multiple IDSs to increase the probability of detecting attacks). In this thesis we will use the term design diversity interchangeably with defence-in-depth and defence-in-breadth unless stated otherwise.

The cost of building a system can be significant for an organisation, and the use of design diversity techniques will inevitably add to this cost. A simple assumption is that the development of N additional variants leads to an increase in cost by a factor of N. While studies on the cost overhead of N-version programming are limited, the general trend of thought is that this assumption is not true. An analysis of a real-world system, developed with two variants over the course of seven years has shown that the cost of a second variant incurred an additional cost of anywhere between 42% and 71% of the cost of building a single variant system [9]. Even so, the added cost may be prohibitive, and over the years an alternative avenue has become more available.

The use of off-the-shelf (OTS) software has become a very popular solution for modern organisations, as it significantly lowers the costs incurred to merely the set up and maintenance of software products, along with required licensing fees. The pool of available OTS systems that offer the same solutions has also increased, and this has led to the possibility of using multiple OTS systems for the purposes of design diversity and increased reliability. Because different OTS products are built from different specifications, this can lead to the products being *functionally diverse*, meaning that they provide the same final results, but do so through different methods and techniques[2]. This offers an extra layer of diversity between the products, while at the same time having their development cost being offset to their respective developers.

A common example of this approach is for the case of intrusion detection systems (IDS), used for network security. The number of available IDS software on the market is quite extensive, with prominent products - *Snort* [36], *Suricata* [37] and *Zeek* (formerly *Bro*) [38] - having been extensively tested. A recent work has looked at the use of design diversity in the context of different rule-based IDS systems [39]. Algaith et. al have also studied the diversity between four different IDS systems (Anomalous Character Distribution, GreenSQL, Apache Scalp and Snort) using different configurations [40]. These products were configured in 2-version systems and used to classify SQL injection attacks on three different web applications. The authors report an average increase in sensitivity (the number of attacks correctly identified) of 60%, with a respective decrease of 10% for specificity (the number of non-attacks correctly identified) in 1-out-of-2 adjudication schemes. For 2-out-of-2 configurations these values were flipped[3].

Another case of design diversity can be the use of multiple antivirus (AV) products. Gashi et. al published a study in 2009 [41] focusing on the diversity between AVs found in the VirusTotal web platform. VirusTotal is a service that allows users to submit file samples for analysis performed by multiple AV products (53 at the time of writing). In the study, the authors collected a total of 1599 malware samples through the use of a distributed honeypot[4], between the months of February and August, 2008. These malware

[2]For example, antivirus systems might be signature-based or behaviour-based. They offer the same result - classifying a file as malware or benign - but do so through different processes and with different levels of success for different types of files.

[3]The adjudication scheme for a design diversity system dictates how the outputs of each variant are used to produce a final output for the overall system. We will discuss the use of different adjudication schemes in Section 2.2.

[4]A honeypot system is a purposefully exposed system used to draw malicious agents to attack. The

samples were sent to the VirusTotal service for a period of 30 consecutive days, and the classifications given by each AV product recorded. The conclusions show that, while no single antivirus product detected all malware samples, 18% of 1-out-of-2 AV pairs achieved a perfect detection rate, and 32% of pairs achieved a better detection rate than the single best AV product individually.

The same authors provided a follow-up study of the same dataset [43], where they analysed the performance gains of using more than pairwise systems. One of the results is that 25% of triplets successfully detected all malware samples. One of the more surprising results found in both of these studies is that of "AV regression". Because the malware samples were sent multiple times to the VirusTotal platform on consecutive days, there are instances of certain AV products failing to detect a sample which they had previously detected. The reasons for these regressions are not clear, but they concern the manner in which AV manufacturers continuously update their products.

Other studies have looked at diversity benefits when using multiple SQL database servers [44], operating systems [45], etc. *Ensemble learning* is also widely used in machine learning [46], which is a set of techniques meant to incorporate several different base models into a single system. It is clear that the use of multiple software versions can lead to improved system reliability and security, sometimes significantly so. The big question here is being able to assess the improvement that can be achieved, and whether the gained reliability is cost-effective. The benefits gained will be dependent on the field they are being applied to and will differ from application to application. For example, Littlewood and Strigini warn that the performance obtained from using multiple IDSs must be measured using attack samples of the same type, rather than an average mixture of attacks, as the use of design diversity will have different impacts when used to tackle different problems [5]. This emphasises the need for empirical research in different fields of study, and the usefulness of design diversity will largely depend on its effects in particular industries.

2.2 Software fault and intrusion tolerance

As we have illustrated before, in section 2.1, the initial step that leads to the failure of a system is a fault[5]. Therefore, increasing the reliability and dependability of a system is mostly focused on the fault stage, and making sure that a fault is not allowed to propagate into an error down the line. There are four major types of solutions when dealing with faults [21]:

- Fault prevention deals with preventing the occurrence or introduction of faults into a system.
- Fault tolerance enhances the ability for a system to resist the occurrence of faults, and stops faults from propagating into errors.
- Fault removal deals with the reduction of the number or severity of faults.

attacks made against these systems are collected and can be used for future analysis. The honeypot system used for this study was SGNET [42]

[5]The attack step specified for security *leads* to a fault, which is the first step that occurs within the boundaries of a given system.

- Fault forecasting is used to estimate the number of faults in a given system, as well as the likelihood of future faults occurring.

All of these techniques are useful in increasing the reliability and dependability of systems. In this report we will mostly be focused on fault tolerance. An early viewpoint on the use of fault tolerance techniques is that it is useful only when dealing with hardware because of physical faults, which are inevitable over time. However, when looking at software and its design, the errors made in it are purely design errors, expressed by human oversight or lack of foresight, and should instead be identified and corrected, rather than tolerated. While this is a desirable objective, in most practical cases it is unattainable due to the complexity of the software that is constructed. Faults are inevitable and if encountered in operation will lead to failures. The reliance on fault tolerance is more pronounced when integrating off-the-shelf components into a wider system, as in some cases the code cannot be changed.

The objective of fault tolerance is to correct the erroneous state of a program before this leads to a failure of the system. An initial approach to this objective is to build additional checks into the system. This is so that we can detect errors in the system in time for them to be corrected, and before they are able to propagate through the system and to different components. This can be done by placing a "checking" unit after an operation, which checks the output generated by a component and either accepts it or rejects it. When rejecting the output, two options are available. A designer can either choose to repeat the operation again, optionally changing the system's state somehow to attempt to mitigate whatever lead to the error, or an error message can be propagated through the system instead, allowing other components who receive it to adapt to the failure of their previous component.

This approach however has a few drawbacks. Firstly, the use of a checker component can lead to a significant increase in response time overhead for the system, which depends on how difficult the checking process is. Furthermore, the checker unit itself represents an additional component of the system, which is also vulnerable to containing faults. Under normal circumstances, it may also not be possible to derive enough tests to cover the possible ways in which a component may fail.

The next logical step then, is the use of design diversity, which we have discussed in the previous section. In the next sub-sections, we will be discussing how one constructs a voting system, which produces a final results based on the result generated by the different variants in a system.

2.2.1 Binary and non-binary systems

Before discussing the different voting options available when using design diversity, it is important that we differentiate between the different types of systems that exist. There are two major types of systems which we can apply design diversity techniques to, binary and non-binary. Binary systems (often referred to as *binary classification systems*) are problems where the outputs of the system can only take one of two values, usually expressed as true or false values. A non-binary system on the other hand, is a system where multiple outputs are possible (for example, a machine learning system which attempts to classify the animal present in an image). These non-binary systems can further be divided into cases where the possible outputs are part of finite or infinite sets.

A classic example of a binary classification system is found in security contexts, where a system is tasked with classifying items of interest as either malicious or benign. For example, an intrusion detection system set to monitor the contents of a network. As there are only two possible classifications (malicious or benign, positive or negative), we can construct a confusion matrix that perfectly describes the problem in question. We show this in Table 1.

Table 1: Binary classification system - confusion matrix

	Actual: Positive	**Actual: Negative**
Classification: True	TP	TN
Classification: False	FP	FN

Vertically, the two columns describe the possible values of the item being analysed (either malicious (positive) or benign (negative)). Horizontally, the two rows describe the outputs generated by the classification system. The two cells highlighted in green are the cases where the system performed its job correctly, i.e. the system correctly identified the items as malicious or benign. These two cases are known as *true positives* and *true negatives*. The red cells represent cases where the system incorrectly classified the items. These are known as *false positives* and *false negatives*.

2.2.2 Adjudication schemes

An adjudication scheme is a scheme by which one decides how to produce a final output based on the outputs produced by the individual variants present in a system. As we have discussed before, a system which uses design diversity includes multiple components that work in parallel to produce the same results. On top of these variants, a system must implement additional code which makes use of the internal results and decides how the system overall should respond.

We will start by looking at the most basic type of design diversity, where we only have two variants working in parallel to solve a problem. Let us imagine two variants, S1 and S2, which are looking at traffic passing through a network, and must decide if each message is malicious or benign. This presents a binary classification system (as described in the previous section), where each variant can generate an alert, or not. This is the scenario we illustrate in Fig. 4.

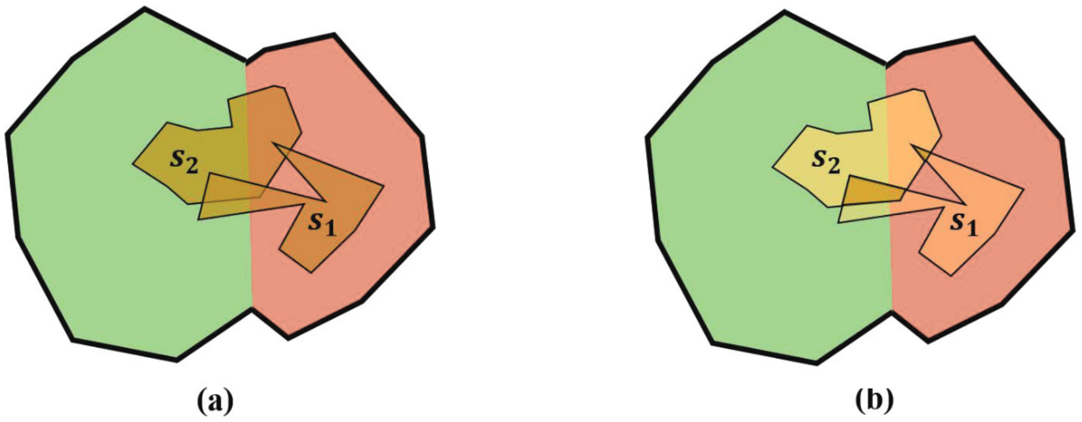


Figure 4: 1-out-of-2 (a) and 2-out-of-2 (b) adjudication schemes [2]

The green and red areas in the figure represent ”demands”, that is, each message passing through the network, with green areas representing benign messages, and red areas representing malicious messages. The S1 and S2 areas in yellow, cover the demands for which the respective variants generated alerts, meaning that areas of S1 and S2 covering red demands represent true positives generated by the variants, and their areas covering green demands represent false positives. Our goal in this system is therefore to adjudicate between the alerts generated by S1 and S2 so as to minimise the number of false positives generated, while maximising the number of true positives.

There are two basic adjudication schemes which we can employ. A 1-out-of-2 (1oo2) adjudication scheme (represented in the figure as (a)) generates an alert when either of the variants generates an alert. This scheme always results in an equal or higher number of overall true positives generated. The second adjudication scheme we can make use of is a 2-out-of-2 (2oo2) scheme (represented in the figure as (b)). A 2oo2 adjudication scheme generates an alert only when both variants agree on generating an alert, and this always leads to an equal or lower number of false positives.

The decision between a 1oo2 and 2oo2 adjudication systems is largely dependent then, on the properties of interest for any particular system, whether a higher number of both true positives and false positives is superior to a higher number of true negatives and false negatives, or vice-versa.

One can then extrapolate these adjudication schemes to scenarios where more than two variants exist. This would lead to R-out-of-N (RooN) adjudication schemes, where we produce a certain result if, and only if, R out of N variants agree. These adjudication schemes are ”consensus” schemes. The survey by Nascimento et. al [17] presents other common adjudication schemes:

- Majority scheme - As the RooN scheme, but now R must be a majority of variants rather than a predefined number.

- Formal consensus or majority schemes - These are the same as the consensus (RooN) and majority schemes, but allow for the variants to be within a certain distance of their results for them to be considered in agreement. This is useful if we are dealing with non-binary classification systems, such as calculating distances, for example.

- Median scheme - Simply the selection of a median result as the final system result.
- Mean and weighted schemes - Simply the selection of the mean or weighted average of the variants.
- Dynamic consensus and majority schemes - Like the consensus (RooN) and majority schemes, but with the addition that the number of variants can change, if certain variants fail to produce a result, or for some other reason are considered to be excluded.

The decision of which adjudication scheme to utilise will depend on the context in which it is applied. Majority schemes tend to strike a balance between the generation of false positives and false negatives, which is useful if both types of failures have similar costs. Formal consensus and dynamic consensus offer similar benefits, which added adaptability when dealing with more intricate outputs. Median and mean schemes are particularly useful when the outputs of systems may be non-binary, taking place between a range of two values.

2.2.3 Optimal adjudication

One adjudication scheme we have not discussed yet is known as an optimal adjudicator scheme [47, 48]. An optimal adjudicator is special because it can only be used after analysing a system using design diversity with a different adjudication scheme.

In order to generate an optimal adjudicator, we must divide the problem into *syndromes*. A syndrome is defined as a unique combination of variant outputs, for example, S1 outputs ”true” and S2 outputs ”false”. This list of syndromes should be exhausted for the problem in question. With the system’s syndromes defined, we can then look at historical data and use an oracle function to label it, such that we can look at the occurrences of each syndrome and determine how many true and false, positives and negatives would be observed, depending on the possible overall outputs for the system. With this knowledge then, it is easy to decide what the final output of the system should be, under each specific syndrome, such that it maximises the desired properties while minimising the costs incurred from incorrect output.

Let us imagine now, as before, a system that is looking at network traffic passing through a network, which needs to be classified into either malicious or benign (1 or 0). Let us add an additional variant, S3, alongside the two already existing. Table 2 shows all the possible syndromes we can make with a 3-version system, where ”1” means the variant generated an alert, and ”0” means it did not. The rows ”positives” and ”negatives” show us the number of malicious and benign messages that were observed under the specific syndromes.

Table 2: Optimal adjudication example - S1,S2,S3

Outputs from S1,S2,S3	**0,0,0**	**1,0,0**	**0,1,0**	**0,0,1**	**1,1,0**	**1,0,1**	**0,1,1**	**1,1,1**
Positives	13	54	43	31	23	12	8	156
Negatives	78	5	45	10	9	17	2	37
Optimal adjudication	0	1	0	1	1	0	1	1

Looking at this table we can easily define under what syndromes an optimal adjudicator should generate an alert (1) or not (0), simply by observing whether there were more positive or negative demands for that syndrome. Because the optimal adjudicator is defined a posteriori, we can easily show that no other adjudication scheme would have performed better[6].

There are other interesting conclusions that an optimal adjudicator might give us. Notice how, in our (somewhat contrived) example, an optimal adjudicator would generate an alert if either S1, or S3 generate alerts individually, but would not generate an alert if both S1 and S3 alerted on the same demand. A situation like this would indicate that the two variants have overlapping "failures", which would not have been detectable with any other adjudication scheme.

An optimal adjudicator has some obvious drawbacks, however. Firstly, because it requires an a posteriori analysis of the problem space, it cannot be used when first implementing a system. Secondly, while an optimal adjudicator works well for a binary classification system, a different problem space that involves finite solutions (such as arithmetic outputs) would likely result in too many possible syndromes to correctly define the system. As such, an exhaustive list, like the one we presented above would likely not be possible.

2.3 Performance assessment for binary classifiers

Binary classification systems' performance is often assessed based on the number of true and false positives and negatives they generate, which we have described previously in Section 2.2.1. In order to assess the performance of a binary system, a confusion matrix like the one in Table 1 is created.

After constructing this confusion matrix, one can then use its values to calculate a myriad of metrics that attempt to capture the performance of a specific tool. In Table 3 we show a number of commonly used assessment metrics for measuring the performance of intrusion detection systems, as described in [3]. The names of the metrics change depending on the literature, and where possible we have identified all names used for each metric. In the next sections we give a brief overview of all of the metrics found in the table.

[6]This is not necessarily always the case, it's possible that it can perform worse based on using incorrect training data. We will touch more on this in a later chapter.

Table 3: List of assessment metrics [3]

Name	**Formula**	**Description**
Precision	$\frac{TP}{TP+FP}$	Proportion of the classified positive cases that are correctly classified. Also referred to as *true positive accuracy*, *positive predictive value* or *confidence*
Sensitivity	$\frac{TP}{P}$	Proportion of positive cases that are correctly classified as positive. Also called *true positive rate* or *recall.*
Specificity	$\frac{TN}{N}$	Rate of negative cases that are correctly classified negative. Also known as *true negative accuracy* or *inverse recall.*
$\mathrm{F_x}$ Score	$(1+x^2)*\frac{sensitivity*precision}{(x^2*precision)+sensitivity}$	Represents the weighted average of the precision and sensitivity, producing values between 0 (worst) and 1 (best). When x >1, this measure weights sensitivity higher than precision, while x <1 does the reverse. This goes by the name of *F-Measure* when x = 1, thus valuing sensitivity and precision the same
Inverse precision	$\frac{TN}{FN+TN}$	Rate of classified negative cases that are indeed negatives. Also known as *true negative accuracy* or *negative predictive value.*
Accuracy	$\frac{TP+TN}{P+N}$	Represents the proportion between the correctly classified cases and the total case.
False positive rate	$\frac{FP}{N}$	Represents the ratio of negatives that are incorrectly classified as positives. Also called *fall-out.*
False negative rate	$\frac{FN}{P}$	Proportion of positives that are incorrectly classified as negative. Also referred to as *miss rate.*
Percentage of wrong classification	$100*\frac{FN+FP}{TP+FN+FP+TN}$	Percentage of total cases that have been incorrectly classified.
False detection rate	$\frac{FP}{FP+TP}$	Represents the ratio of reported positives that were incorrectly classified. IN some cases it is incorrectly used unded the name of *false positive rate.*
Informedness	$sensitivity+specificity-1$	Quantifies how consistently the predictor predicts the outcome, i.e. how informed a predictor is for the specified condition, versus chance.
Markedness	$precision+inverseprecision-1$	Quantifies how consistently the outcome has the predictor as a marker, i.e. how marked a condition is for the specified predictor, versus chance.
Matthews correlation	$\frac{TP*TN-FP*FN}{\sqrt{(TP+FP)*(TP+FN)*(TN+FP)*(TN+FN)}}$	Represents a correlation coefficient between the true classes and the classified results. It is also equivalent to the geometric mean of *markedness* and *informedness.*

2.3.1 Sensitivity and Specificity, Precision and inverse precision

The metrics of sensitivity and specificity, initially coined by Jacob Yerushalmy in 1947 with the intent to assess methods of medical diagnosis [49], are presently two of the most commonly used metrics for measuring the performance of intrusion detection systems.

Sensitivity (also called *Recall*, *True positive rate* or *Probability of detection*) represents the proportion of true positives which were correctly classified as such by the system being measured. On the other hand, specificity (also called *True negative rate* or *Inverse recall*) represents the proportion of true negatives which were correctly classified as false.

Both of these measures can be calculated using the following formulas:

$$Sensitivity = \frac{TP}{P} \tag{1}$$

$$Specificity = \frac{TN}{N} \tag{2}$$

The importance of these two metrics is dependent on the circumstances in which they are being applied. In the medical field, for example, a highly sensitive medical diagnosis, one which rarely misses the condition being tested for, is a very important trait, even if it turns out later that it generates a large number of false positives. Other scenarios, such as facial recognition for security (unlocking a phone through facial recognition), specificity can be a more important trait so as to not give access to unauthorised agents.

Both sensitivity and specificity are affected by skew in the number of actual positives or negatives in the classification space. If the number of positives or negatives is low, then we will have little confidence in these metrics. In a similar note, if the number of positives is significantly different than the number of negatives, then the values of sensitivity and specificity may not be giving the same level of information.

While sensitivity and specificity measure the system by the actual positive and negative rule (the denominator in their equations is the actual number of positives or negatives), precision and inverse precision measure the system by the predicted positive and negative rule. In other words, precision represents the proportion of positive classifications that were correct, while inverse precision represents the proportion of negative classifications that were correct.

Both of these measures can be calculated using the following formulas:

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

$$Inverse\ precision = \frac{TN}{FN + TN} \tag{4}$$

2.3.2 F-x Score

F-x score represents the weighted average of the precision and sensitivity, and allows the experimenter to place a higher value on either precision or sensitivity, according to the needs of the context which is being applied.

F-x score is calculated using the following formula:

$$Fx = (1 + x^2) * \frac{sensitivity * precision}{(x^2 * precision) + sensitivity} \tag{5}$$

The values of F-x vary between 0 (worst) and 1 (best), and the value of x is used to change the weight of either sensitivity or precision. When x >1, sensitivity is weighted higher, while when x <1, precision is weighted higher. A special case occurs when x = 1, which represents the harmonic mean of sensitivity and precision.

2.3.3 Accuracy and percentage of wrong classification

Accuracy is a measure that captures all of the values present in the confusion matrix, and represents the proportion between the correctly classified cases and the total case. Essentially, accuracy represents the total number of cases which were correctly classified by the classifier. Inversely, the percentage of wrong classification is the proportion of cases which were incorrectly classified.

Accuracy and percentage of wrong classification are calculated using the following formula:

$$Accuracy = \frac{TP + TN}{P + N} \tag{6}$$

$$Percentage\ of\ wrong\ classification = 100 * \frac{FN + FP}{TP + FN + FP + TN} \tag{7}$$

Special care needs to be taken when utilising accuracy and percentage of wrong classification to measure the performance of a system. Often, the value of true positives and true negatives are not the same when analysing a particular context, and one will often be more important than the other. By using both values for the same measure, this can become "tainted" and lead to misleading results.

2.3.4 False positive rate and false negative rate

The false positive rate and false negative rate are the proportions of negative cases which yielded positive classifications, and positive cases which yielded negative classifications, respectively. False positive rate is also sometimes referred to as *Fall-out*, and false negative rate is sometimes called *Miss rate.*

False positive rate and false negative rate are calculated using the following formulas:

$$False\ positive\ rate = \frac{FP}{N} \tag{8}$$

$$False\ negative\ rate = \frac{FN}{P} \tag{9}$$

2.3.5 Informedness, Markedness and Matthew's Correlation

Informedness, also called *Youden's J statistic*, is a way to estimate the probability of an informed decision, and its values range between 0 (zero) and 1. A value of 0 (zero) would mean that the classifier generates the same number of positive classifications independently of the number of actual positives in the dataset, i.e. the classifier would be random, whereas a value of 1 indicates that there are no incorrect classifications generated by the classifier. Informedness is often used in conjunction with a ROC curve. Fig. 5 shows how informedness (J) represents the maximum distance between the chance line (dashed) and a classifier's line.

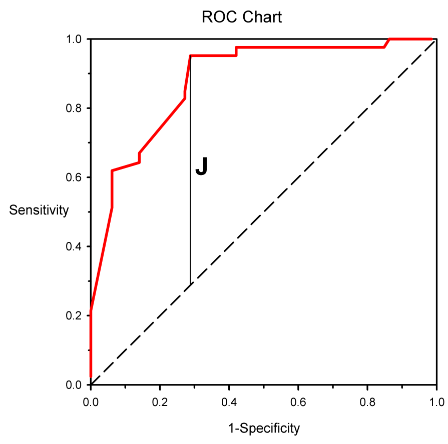


Figure 5: Informedness represented in a ROC curve

Informedness is calculated using the following formula:

$$informedness = sensitivity + specificity - 1 \tag{10}$$

Markedness is calculated using the following formula:

$$markedness = precision + inverse\ precision - 1 \tag{11}$$

Matthews correlation was originally introduced by B. W. Matthews in 1975 for the field of biochemistry [50]. More recently, it has been adopted in the field of machine learning, as a measure of the quality of binary classifications. It takes into account true positives, true negatives, false positives and false negatives, and can be used even if the classes are of very different sizes. This aspect of being capable of handling different sized class sets is generally regarded as one of the best ways of avoiding biases found in other, more commonly used metrics [51].

Matthew's correlation takes a value between -1 and +1, where a value of +1 represents a perfect classification, 0 represents no better than random prediction and -1 represents the opposite of a perfect classification (all incorrect classifications).

Matthew's correlation is calculated using the following formula:

$$Matthew's\ correlation = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP) * (TP + FN) * (TN + FP) * (TN + FN)}} \tag{12}$$

2.4 Malware

The threats of malware - software designed to induce faults in other software systems - is certainly not a new thought, with the idea of a computer virus having existed in fiction, even before they were a reality [52]. The coinage of the term "virus" in the context of computer security is largely attributed to Frederick Cohen in 1987 [53], where he describes a computer virus as "a program that can infect other programs by modifying them to include a possibly evolved copy of itself".

As the author of the book Computer Viruses and Malware points out [54], "...there is no universally accepted definition of terms like 'virus' and 'worm'...", and as such it's more important that we take into account the various characteristics associated with a specific malware sample. There are two characteristics that are particularly important to understanding the propagation of a piece of malware: whether or not the malware is self-replicating, meaning that it copies itself and attempts to infect more machines; and whether or not the malware is *parasitic*, meaning that it needs another piece of software in order to be executed.

What is typically described as a virus normally encapsulates both of the above characteristics. Viruses self-replicate into existing executable code in order to propagate themselves, normally in order to infect a network of computers. It's important to note that requiring being embedded in executable code, and what that entails, is somewhat loose. An executable code could be, in the strictest sense, intentionally injecting malicious lines of code into an otherwise benign piece of software, but it could also refer to interpreted code, for example, "macros" in a text processing software.

An example of a malware which is self-replicating but not parasitic could be a worm[7], which is a stand-alone malware that requires an automated or user-initiated launch to be executed. These are typically propagated through email with phishing attacks, or through more direct file transfer. For worms, there isn't even a necessity that they perform malicious activities beyond propagating, as its self-replicating nature could be used solely as a means of flooding a network. On the other hand, a malware could be parasitic but not self-replicating. Logic bombs are normally embedded directly into the code of other software, meant only to trigger once a certain condition is met (such as a specific date and time, a user login, etc.). Logic bombs are of particular concern in regard to smartphone applications, especially Android devices, where rogue developers may obfuscate their malicious actions in the scope of a larger, benign looking application [55]. Trojan horses are also part of this category, and often act by opening back doors for other, more sophisticated attacks, such as by opening certain ports on a computer network.

These three distinctions we have made above largely deal with a malware's life cycle and the way in which it propagates. It is also important to note two other aspects of a malware's nature: their infection vectors, and their payloads. In terms of a malware's infection vector, this can include direct file transfers, for example via USB drives, discs or directly connected devices; email transfers, through email client auto-run settings or phishing attempts; direct download from untrusted or unsecured connections. Due to the

[7]There is little consensus as to the categorisation of malware and no concrete definition of what a 'worm' is. As such, other works may disagree with our classification of a 'worm' as a self-replicating but not parasitic malware.

vast number of ways in which malware can be propagated, it is difficult to confidently defend against all points of entry into a system or network, especially when many of these weak spots are due to human errors, such as being tricked into performing an action through social engineering, or through sheer negligence.

Categorising a malware sample based on its payload (its intended goal) is somewhat more straightforward, as these categorises are easily differentiated between themselves. Broadly, malware can be categorised into three types based on its payload:

- Rootkits allow attackers unauthorised access to a system or network and can be used to perform a variety of attacks, such as gathering information (spyware), tampering with the systems, or outright perform destructive actions such as deleting entire databases.
- Ransomware malware attacks a victim's system by encrypting as many files and programs as possible, before demanding a ransom (typically paid in bitcoins or other digital currency) in return for a decryption key.
- Spamware or adware represent more tame styles of malware, which generally focus on the injection of advertisement into otherwise innocuous software or onto the operating system.

2.4.1 Malware detection

Previously, we have briefly discussed the idea of intrusion detection systems (IDSs) and their prevalence in protecting software systems. The CIDF (common intrusion detection framework) is a working group started by DARPA in 1998, with the goal of coordinating and defining a common framework in the IDS field. Their proposal for a general IDS system follows as Fig. 6, with four distinct types of functional modules:

- Event modules, which are composed of the sensor elements that monitor the desired environment.
- Database modules, which store event information captured, as well as subsequent information generated by the other two modules.
- Analysis modules, which are used for processing the various events captured and detect potential intrusions.
- Response modules, which are responsible for responding to a detected intrusion.

Intrusion detection systems are usually divided into two distinct categories, based on the level at which they operate at - network-based IDSs and host-based IDSs. Furthermore, IDSs can be categorised based on their method of operation. The first of these is signature-based methodologies, which revolve around the generation of *signatures* that uniquely identify samples of code, mostly through direct code analysis, and compare these to known malicious signatures. The other approach is known as behavioural-based detection, or anomaly-based detection, in which the data and activity produced by a piece of software is analysed for potential malicious actions.

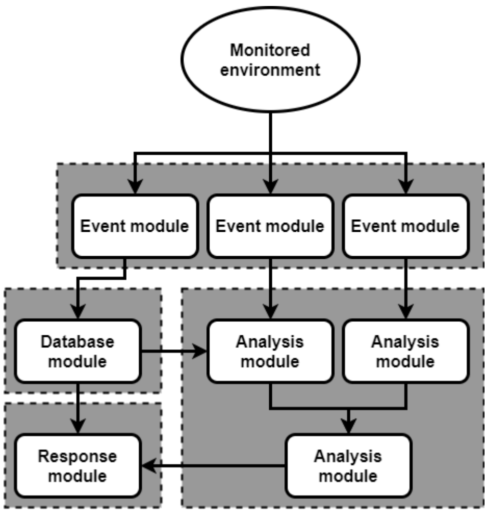


Figure 6: Common intrusion detection framework

2.4.2 Signature-based detection

Signature-based detection methods rely on the generation of signatures that uniquely identify a piece of software or activity pattern. There are many different ways to uniquely identify such samples, such as the extraction of byte sequences from software samples, the collection of metadata from host machines, and the traffic patterns of activity within a network, among others. A signature, once generated for a particular intended sample can then compared with a database of known malicious signatures, and an alert then launched if a match is found. These signatures can be as simple as looking up the IP address from a given incoming packet, or as complicated as finding specific bytecode sequences of executable files. Static code analysis is a subset of signature-based detection methods, which are mostly used to test source code and detect faults left by developers, but also have some application in detecting malware. Specifically, they can be used to detect logic bombs or trojans left *intentionally* in source code by a rogue developer.

These techniques are often used as an initial front of defence for a system or network. Because generating a signature for a particular sample is quick, and the lookup of a known malicious database is computationally simple, these methods are often used as an initial filter, to detect the most basic of malicious samples which would otherwise be costly to analyse in further defence steps taken by a system.

While signature-based detection methods work great as an initial, fast step of detecting malicious samples or activity, they are far from being the most versatile way of defence. Their main drawback is that, in order to detect malicious activity, a detection system needs to already have knowledge of malicious signatures in its internal database. Compiling these databases can be difficult, however, open-source intelligence (OSINT) sources can be used for this aspect. A problem which is not so easily overcome is that attackers can often slightly change their activity patterns in order to avoid detection. When dealing with malware detection, these methods are often ineffective at dealing with samples that make use of code obfuscation techniques [56]. Such malware is known as polymorphic, and works by incorporating a replication engine in its own code which generates functionally identical, but differently coded versions of itself. More sophisticated signature-based detection systems can sometimes deal with code obfuscation, such as by looking at system API calls that do not necessarily change when source code is altered [19]. However, these cases also rely on the malware not being sophisticated enough to obfuscate their own actions and activity patterns. For such malware, behaviour-based detection methods are more desirable.

2.4.3 Behaviour-based detection

Behaviour-based detection techniques, also known as anomaly-based techniques function by analysing the activity of suspicious content and detecting when this deviates from a pre-defined "normal" activity. The need to define what "normal" behaviour looks like, as well as to keep updating this notion as a system evolves introduces a large cost and complexity to these malware detection systems. However, paying such drawbacks also allows these systems to achieve a number of very important benefits, one of the biggest being that behaviour-based detection systems are able to detect brand new attacks and malware. There are three major ways to achieve a behaviour-based detection system [57].

In **statistical-based approaches** a sample of the activity in question is generated and from it a "normal" profile is generated. This can take multiple forms. For network intrusion detection, factors such as the rate of traffic flowing through the network or between certain hosts, the number of distinct hosts communicating, the various protocols or metadata about packets being observed can be used to generate this "normal" profile. For host-based detection, the activity generated by a sample can be taken into account, such as the type of API calls made to the system, the number and mode of access to the disk, etc. When such a detection system is then put into production, new activity is compared against the previously generated normal, and if these are sufficiently different then an alert is raised. The notion of the "normal" profile of activity can and should be continuously updated as the detection system, and the environment it protects, evolve over time. In this sense, the generation of a "normal" profile is easy, as this is an automated process conducted by the detection system itself, however, this also leads the system open to be "trained" by attackers, looking to shift the idea of what a "normal" profile looks like.

Knowledge-based approaches work somewhat differently, by utilising the notion of *rules* to determine whether a particular sample is malicious or not. These rules attempt to uniquely make the distinction between what is a benign sample and what is a malicious sample, through the identification of specific data points. In certain circumstances, these rules are even constructed by a human expert. A major benefit of this approach compared to a statistical based one is that the number of false positives is reduced. This is because in statistical approaches, anything that deviates from the "normal" profile is regarded as malicious, even if it is indeed benign, but had simply not been observed in the past. However, a major drawback is the requirement for knowledge of a system, such that creating the necessary rules is feasible.

Finally, **machine learning-based approaches** function akin to statistical approaches, by establishing a model that allows for the categorisation of sample activity into either normal or abnormal. The use of machine learning techniques, however, allows for the ability of the system to learn as it acquires new information, rather than relying solely on the basis of previous results. However, the use of this approach is held back by the expensive nature of the building and using of machine learning models. Still, machine learning approaches allow for the highest amount of flexibility and adaptability by the part of the IDS system.

2.4.4 Machine learning and malware detection

One of the more researched aspects of behaviour-based detection methods nowadays are those involving the use of machine learning algorithms. A major proponent for this shift in interest towards machine learning is that the high rate of malware spread on the internet has compounded to the point that manually-maintained and evolved defence systems are no longer appropriate [58]. The key point on the use of machine learning algorithms is that these make use of past experience in order to improve their performance and make more accurate predictions [59], and this allows organisations to efficiently keep up with a constant barrage of new malware and attack attempts.

Constructing a machine learning model consists of three major steps [60]: **i.)** the extraction of features from the input samples to be analysed - these will vary depending on the type of samples that require classification, such as network traffic, executable files, etc.; **ii.)** feature selection, which involves deciding which of the extracted features are necessary for classification, and which ones are redundant for the task at hand - this reduces complexity and ultimately speeds the process of classification; and **iii.)** the injection of the selected features into a chosen machine learning classifier so that it can learn the distinction between benign and malicious samples, and be able to accurately classify newly seen samples.

The features used by machine learning algorithms are much the same as used by other malware detection techniques, such as bytecode and binary sequences, and network and host activity. Many of these features are unnecessary for the classification process. However, making the distinction between which features are useful, and which ones merely add complexity to the classification process is not an easy task to be operated manually. This is another aspect that the use of machine learning algorithms is significantly more efficient than a manual approach.

The choice of machine learning classifiers for malware detection has been quite extensively researched. One of the most well-known machine learning classifiers is known as a **support vector machine (SVM)** [61–63]. These classifiers work by essentially determining the hyper-plane that separates the sample data into two categories, benign and malicious, with respect to the multi-dimensional feature vector being analysed. **Random forests** [63, 64] are another popular classifier, and work by employing an *ensemble* of various decision trees. The number of trees, their length and complexity at each set of nodes gives them much the same qualities achieved with the use of design diversity, and the output of these methods is usually the class most chosen amongst all of the trees utilised. A more recent approach are **neural network** classifiers, which are constructed from interconnected nodes. An initial layer of nodes is used for input to the model, which is then transferred into a variable number of hidden node layers that process the input into something that can be outputted in a final output node layer. Different variations of neural networks exist, such as *recurrent neural networks (RNN)* [65] and *convolutional neural networks (CNN)* [66]. The combination of more than one of these classifiers is also a possibility, increasing the complexity of the system but potentially increasing its accuracy as well [67].

2.4.5 Malware repositories

Malware repositories are a necessary part of the development of any malware detection system. A behaviour or machine learning based approach requires the use of malware samples for training purposes, and any other approach ought to have its performance assessed by testing it against a realistic set of samples (both benign and malicious). Creating malware repositories is difficult, and requires that it is regularly updated so as to accurately represent the current landscape of malware. The motivation for this is the continuous discovery of new malware strains, which saw upwards of over 200,000 never-before-seen malware variants in the first half of 2020 alone, according to a report by SonicWall [68]. It's also majorly important that the malware kept in these repositories is sufficiently varied so as to account for the various different types of attacks that a tool would be employed to defend against.

Malware repositories are often populated through the use of honey-pot networks. These are networks that are specifically set up such as to appear vulnerable to would-be attackers on the internet. Any attack carried out against a honey-pot infrastructure is then captured and catalogued in a repository. Malware repositories may also be populated through manual reverse engineering of attacks [69], which is generally the only way to include major, specifically targeted malware samples in a repository, such as the malware used in the Stuxnet attack [70].

VirusTotal and other similar tools are an important aspect for the creation of repositories. VirusTotal makes use of a large number of individual antivirus tools to classify software samples as benign or malicious, and provides a trustworthy first check for the true nature of a software sample before it is included in a repository. Additionally, for samples deemed as malware, VirusTotal also generates a type classification for the sample in question, allowing repositories to be categorised based on types of malware, a feature which is important when assessing detection systems.

2.5 Web application threats

The increased growth of online platforms, traffic and services provided over the years has had significant impacts on the types of attacks created each year. Symantec reports a total of 246 million new malware variants during the year of 2018 [10]. The global COVID-19 pandemic has forced many organisations and individuals to place more emphasis on web technologies for remote working, and this has also lead to an increase in cybercrime looking to take advantage of less than prepared companies [71, 72].

The OWASP foundation [73] - an organisation dedicated to the improvement of web application security - catalogues the most commonly exploited vulnerabilities for websites. The foundation classifies injection threats has the number one most prevalent threat to web application security [74], a spot which it has occupied since 2010. Injection exploits have been a constant attack vector in the web application landscape [75], and refer to a class of attacks where data sent from the client to the server is not properly sanitised before being processed. The most common example of these attacks are SQL injections, where data taken from user input boxes is used to create SQL queries to be sent to a back-end database. If the text supplied by the client is not sanitised, a malicious actor can incorporate valid SQL code into their input such as to bypass or overwrite the intended use of the original query. This can be done to bypass login systems, extracting data from a database, or otherwise manipulating it, such as deleting, editing or inserting malicious content [76]. Injection attack vectors are often easy to protect against, as a variety of different tools and techniques can be used to detect vulnerabilities - such as penetration tools [77] and static code analysis [78, 79]. However, injection vulnerabilities are still prevalent, often attributed to bad coding practices by developers.

While injection flaws are targeted towards the server, web attacks can also be targeted at the end-users of websites. A common attack vector when targeting end-users is known as Cross-site scripting (XSS) [80]. Cross-site scripting refers to a class of attacks where end-users are supplied with a tampered version of the website they are interacting with, often containing malicious snippets of code meant to run on the users' computer, mainly in the form of JavaScript. The most basic version, known as *reflected cross-site scripting* involves creating a tampered version of a real web link which is then supplied to an unsuspecting victim. When a user inadvertently visits the tampered link, whatever malicious code was added to it will run on the victim's browser, and in the context of the visited website, meaning that it has access to all of the information on the webpage. For example, a banking website's link could be tampered such that a malicious version would extract the user's authentication cookie and send it to an attacker. Another version of XSS attacks is that of *stored cross-site scripting*, where malicious content is inadvertently stored on a websites' database, for example through the submission of blog posts or comments on a social-media post. When this malicious content is then shown to a regular user, it will be run on the user's browser, if it was not properly sanitised [81].

XSS attacks can have severe impacts, as they can be used perform a variety of different malicious actions, such as impersonating the victim user, carry out any action the user would be able to normally, read data that the user has access and intercept any input made by the user. As with injection flaws, the various forms of XSS attacks are often detectable with the use of penetration testing and code analysis techniques [82].

A major cause of the vulnerabilities we have talked about stem from bad coding and development practices, such as reliance of client-side checks, revealing error messages and stack traces being sent to the client and improper website configurations (for example, using default credentials). There is a vast array of vulnerability scanning tools for web applications [83], both commercial and open-source, that can be used to detect and correct bad practices. However, some web application threats are not dependant on vulnerabilities in the code, and must instead be actively defended against, such as phishing and spear-phishing attempts.

Another common attack employed against web servers are denial-of-service (DoS) attacks. DoS involve the use of multiple, often hijacked computers sending massive amounts of traffic towards a specific web server. The number of requests made is meant to overwhelm the servers, leaving them unable to properly cater to real users, and often times crashing the servers all together. DoS attacks can have a very large impact on web services, particularly those in the commercial space. A web store being taken offline due to a DoS attack will undoubtedly see a decrease in their profits, while at the same time competitors' websites may end up seeing an increase in demand due to the loss of competition. Protection against denial-of-service attacks often comes in the form of external providers with access to additional physical support (servers, etc) that can be leveraged to support when the number of incoming traffic to a webserver greatly increases.

The landscape of online cyber threats is constantly morphing, with different attack types gaining popularity every year. Until recently, the practice of cryptojacking was the dominating force in the field of cyber threats, due to the extreme price inflation observed for bitcoin and other crypto currencies. Cryptojacking consists of high-jacking personal computers and forcing CPU utilisation for mining operations. This could also be expanded upon if an attacker managed to gain access to an online website and was able to inject malicious code, which could force visiting users to mine bitcoin while the page was open in their browsers.

In the last few years however, cryptojacking has somewhat died down and been surpassed by the practice of formjacking [10]. With formjacking, malicious agents inject code into online vendors' websites, which steals users' payment information upon checking out. These are often hard to spot, as the malicious code simply sends a copy of the users' details to the attackers' website while continuing to allow the original transaction to go through, making the stealing invisible to the user.

2.5.1 Web scraping and malicious web scrapping

One of the more relevant web application threat to our research is that of web scraping. Web scraping is the process of using bots to extract content and data from a website [84]. Using the extracted data, the scraper can then replicate entire website contents elsewhere. Legitimately, web scrapping is used in a variety of online businesses which rely on data harvesting. These can include:

- Search engine bots, that crawl and analyse the contents of websites in order to rank them and present them in search results (e.g., web search engines) [85].
- Price comparison websites, which deploy bots to automatically fetch price and product information for the purpose of data aggregation e.g., flight ticket sellers, hotel discovery websites, insurance comparison [86].
- Job listing websites, which employ bots to find and collect job offers from multiple other websites and list them together in one place [87].
- Market research companies, which crawl social media websites and make a profile of consumer behaviour and opinion [88].

Some instances of web scraping bots are essential to the design of the modern web, one of the biggest examples of which being the crawler bots deployed by search engines. Without the use of web scraping technology such functionality would be severely limited or simply not exist on the web. While legally, web scrapping is generally not against the law[8], it can be used for malicious purposes and have real negative impacts for companies and organisations throughout different industries. A common example of malicious web scraping is the scraping of copyrighted content, such as images, texts or other content, in order to be displayed in some form on the perpetrator's website or business. By doing this, the perpetrator can attract traffic to their website, while simultaneously diverting users which would originally have gone to the primary source. This is often used to generate ad revenue, by having a large amount of "free" content generating traffic to the perpetrator's website.

Another use of malicious web scraping is known as price scraping. In price scraping, an attacker launches a series of scraper bots against retail websites, to scrape product information, such as prices and availability. The perpetrator can then make use of this information to undercut their rival businesses, thus boosting their own sales. Because other services exist which allow users to search over all retailer websites for the best price, the necessity to advertise the lowest price is immense, which has led to this strategy becoming widespread in certain industries - a good example being ticket sellers (flights, concerts, hotels, etc). This concept of price scraping is especially important for our research, as we discuss in Section 4 our analysis of datasets which are focused on this activity.

When deciding whether a web scraping bot is malicious or not, an initial point to look at is whether or not the bot adheres to the specifications set out in *robots.txt*. *robots.txt* is part of the widely used Robots Exclusion Protocol web design pattern, where website administrators make a publicly available *robots.txt* which specifies which areas of the website a robot is allowed to crawl during its operation, or what information the crawler is allowed to process. However, the existence of the *robots.txt* file does not prevent the practices of malicious web scraping. Additionally, because this is not an official standard, many well-known and "reputable" crawlers still fail to adhere to it, and thus can appear to be malicious themselves [89].

While the job of all scraper bots, either legitimate or malicious, is the same - to extract website data - the difficult thing for a business to do is not to distinguish between the two, but to actually detect the malicious ones. Legitimate scraping bots are open and in plain sight. They identify themselves and the organisation for which they scrape data for and, often times, if they are deployed by a high profile entity, they are also whitelisted by the websites they crawl, through their IP address. On the other hand, malicious scraper bots don't wish to be detected and risk being blocked from accessing their targeted website, and so they masquerade as normal human users, which no website will want to block access to. For these cases, victim websites need to employ more advanced detection techniques in order to flush malicious web scrapers from their human audience.

2.5.2 Techniques for detecting malicious web scraping

Web scraping activity has become a common sight in the web landscape. A study from 2016 [90] reports that the amount of bot traffic on the internet accounts for roughly 51.8% of all the traffic captured on 100,000 randomly selected domains, more than half

[8]We use the word "generally" here because there are some instances where the use web scraping techniques, while not illegal themselves, may be used for illegal purposes. One such examples is the mass compilation of user information, scraped without user consent.

of which was considered to originate from "bad bots". This prevalence of bot activity necessitates the development and deployment of web scraping detection techniques if an organisation wishes to effectively protect their system. There are a variety of commonly used techniques for detecting web scraping activity. The work done by Alnoamany et al [91] gives an overview of 14 different detection techniques, with different levels of complexity and success. These different techniques can broadly be classified into four distinct types.

Firstly, there are **syntactical log analysis** techniques, which revolve around the review of meta data of incoming requests, such as IP addresses, user-agent and other similar data fields. Information such as this, which can be used to loosely identify incoming requests can be compared against databases of known violators in a rather naive approach. These approaches appear as a common component of most web scraping detector tools, as an initial source of detection, due to fact that these sanity checks are often computationally expedient. The use of known violator databases allows for an extra ability of sharing security information amongst different organisations and improving overall system security. However, the utility of such basic detection techniques mean that they are easily bypassed by malicious actors. While using syntactical log analysis techniques can easily detect well behaved robots which identify themselves as such, it is trivial for a malicious agent to change the meta information that it sends in its requests, or to change or spoof it's IP address - it is easy for a bot to mask its IP address [92] (for example, through the use of a VPN service). As such, for more critical systems, these techniques are seldom used alone.

A more robust set of techniques are based on **traffic pattern analysis**. Similarly to syntactical techniques, traffic pattern approaches utilise the same access log information about requests made to a server to identify potential robots. However, these analyses are more dynamic than the superficial look at individual data fields, and can focus on more than one request. The most basic version is one that considers the type of content being requested. A study carried out by Geens [93] sets out a simple set of rules for detecting web robots, one of which reading as follows: *the referring field is unassigned AND no images are requested.* By considering the type of content being requested, and comparing it to how it is being requested, it's possible to find clear differences between how a human would make the request vs. how a robot would make the request.

Going a step further, we can group multiple requests into sessions, based on the incoming IP address and user-agent field, within a fixed time period. The patterns found within these sessions can then be analysed for robot activity. Guo et al [94] use these ideas of grouping requests into sessions to develop two styles of detectors which look at the types of resources being requested. The first looks for sessions which requests a single type of resource from a web domain and marks them as "robot candidates". It then compares multiple sessions of single resource type requests for signs of having originated from the same agent. This is a good indicator of a robot continuously polling the same resources to look for changes[9]. The second approach used in the paper considers the rates at which resources are requested and further considers whether or not there are follow up requests for resources embedded in the original resource requested (e.g., images in a web-page).

Another example involves looking for signs of botnets. Carrying out scraping attacks

[9]It is possible for human agents to have similar looking sessions, where a single type of resource is requested, if for example they know its location and navigate to it manually. The paper further defines some measures to counteract these false positives

takes a significant amount of effort, and because attackers often lack the resources, the use of botnets is a prominent solution. One way of detecting the presence of botnets is by looking at the incoming connections from multiple clients and discovering patterns of use or signatures of attack. Acarali et al. [18] detail several different techniques which can be used to detect the presence of botnets (such as looking at similar request sizes, periodical sessions, etc.).

A popular trend for detection techniques is that of machine learning and probabilistic approaches. These are defined as **analytical learning techniques** and describe the more dynamic and adaptable type of detection techniques. Like with traffic pattern analysis, requests are first grouped into sessions. From there different approaches can be applied. Tan et al. [95] propose the use of a C4.5 decision tree algorithm using 25 different attributes derived from the web sessions to classify the requests as originating from humans or robots.

Lastly, a final type of approach for detecting web robots is that of **turing techniques**. These emulate the traditional Turing test [96] to distinguish between human and robot. A major difference between the three previous techniques we discussed and turing techniques, is that turing techniques offer real-time detection, while the previous ones are almost solely exclusive to "offline" detection after the fact. One of the more popular approaches involving turing tests are CAPTCHAs [97], which force the user to answer a non-trivial (for a robot) question, before being allowed to continue with their session. These typically take the form of asking the user to identify text or images.

An alternative to explicit turing tests, such as CAPTCHAs are implicit turing tests, where a user may not be aware that they are being tested. Often this is done through JavaScript tests sent to the client upon a request. These scripts can look for user action, such as mouse movement or clicks, and infer whether they are made by humans or robots through a variety of different ways. An example of this can be found in the work done by Park et al. [98]. In the authors' approach a random value is generated when a user first requests a page. This value is then paired with the resource requested and the IP address of the client, and stored. Alongside the response sent to the user, custom JavaScript is sent which listens for mouse movements from the client, and upon doing so sends back a request with the same value previously generated. If the stored value matches the value of the subsequent request sent via the custom JavaScript don't match, then the session is labelled as a robot. One downside to using such approaches is that not always do users have JavaScript enabled, and this should be considered when deciding to make use of them.

2.6 Cyber threat intelligence and open-source intelligence

An important aspect to the security of any computer system against malicious attacks is the knowledge of the different attack types and techniques that exist and can pose a threat. This knowledge is known as *cyber threat intelligence* (CTI) and often comes in the form of news feeds which an organisation can tap into and make use of to augment their systems. The simplest example of a CTI item is what is known as *blacklisted IP lists.* Blacklisted IP lists are simple lists of IP addresses, subnets or ISPs which have been recognised as malicious sources in the past. These lists can be used to automatically restrict access to certain agents attempting to communicate with the system being protected. Blacklisted IP lists are an integral part of web security, including the detection of malicious web scraping activity.

Another example of CTI comes from the use of VirusTotal [99], and similar tools. VirusTotal allows users to submit software samples for scanning by a large number of different antivirus tools with the aggregate results being then returned to the user. Services like this combine the concept of design diversity to generate rich CTI that can be used not only to confirm the classification of certain software samples as either malicious or benign, but can also allow for a deeper classification of malware by type, which is useful for detailed analysis on the capabilities of other detection tools based on the types of malware they are tested against.

Other types of CTIs exist, with two major ways of obtaining them. The first is to purchase a curated news feed from a company specialised in creating them, such as SenseCy [100] and SurfWatch [101]. These companies provide a subscription-based, up-to-date security news feed containing security threats and information, such as the already mentioned IP reputations lists, malware signatures and indicators of compromise (IoC).

The alternative to purchasing a curated news feed is known as *open-source intelligence* (OSINT), which is the use of cyber threat information which is publicly available on the web. While there are free curated news feeds available online, such as the Cisco Security Advisory [102] and Threatpost [103], OSINT data is usually collected manually from diverse online sources. These sources can be divided into three main categories:

- Structured data sources include information which are sourced in clear, well-defined formats. These include the data obtained from the curated news feeds such as Threatpost, as well as more static information such as those found in vulnerability databases (e.g., NIST's vulnerability database [104]). Purchased curated feeds would also be encompassed in this category.

- Unstructured data sources provide information where the content is presented in a free text format. An example would be that of Twitter, where security information exists but does not follow a standard format, with each security-based account broadcasting security information in whatever format they choose.

- Dark web sources function similarly to unstructured data sources, however, they are generally more difficult to discover. These include information posted on forums and marketplaces on the dark web, and require more evolved dark web scrapers to be able to access and collect.

The collection of OSINT data presents a major obstacle due to how scattered the information can be on the web. Because OSINT sources are often unstructured, finding reliable sources of information presents a difficult task. A large number of research efforts have been dedicated to the use of twitter as a source of OSINT information [105, 106]. Campiolo et al. identified a large number of twitter accounts which report security related events and information, and scraped their twitted messages [107]. They report that a total of 60% of tweets collected were considered important alerts, and that 43% were tweeted before the security event they relay was published in other news websites.

The second issue with the use of OSINT information is the processing of the information gathered, and the assessment of the usefulness of said OSINT data. In order to act on the information gathered one must parse the information from a usually unstructured format, into one that can be acted upon and shared between multiple security systems. The use of machine learning techniques is of particular use in this step. A common use is that of natural language processing, where security information can be extracted from unstructured data sources (such as tweets or news articles) and transformed into structured data that can then be used by security systems [108]. OSINT data, when structured, is known as Indicators of Compromise (IoC), and can be used to enhance the knowledge of security systems and shared amongst different organisations. There is a great deal of value in the capability of sharing these IoCs between different organisations, as it can help in mitigating the occurrence of repeated attacks against multiple organisations. The MISP software [109] is a software which allows for just this, and its use can greatly increase the speed at which newly discovered attacks are known to organisations, and proper defences can be mounted.

OSINT data and IoCs are of particular relevance to SOC teams. They can be integrated into SIEM systems, allowing visibility for security administrators of the current threats posed against their systems, and better identifying and classifying the security events that happen throughout the network.

2.7 Conclusions

In our literature review we have discussed the topic of software reliability and security, including different techniques of increasing these - N-version programming and design diversity. We have presented a number of different metrics which can be used to assess the performance of binary classifiers, which we will make use of in our studies herein. Additionally, we have given an overview of malware and malware detection techniques, using signature and behaviour-based approaches, with an additional focus on web application threats, include malicious web scraping and its detection. Finally, we touched on the topic of open-source intelligence and its usefulness for improving the security of systems and the potential of research work.

3 Methodology

In this chapter we give a brief description of the methodology we followed in order to carry out our analysis, and how this methodology can be used in other contexts which we have not included in our own work. More in-depth information about the methodology of each body of work can be found in the respective chapters. Our work focuses on the assessment of binary classifier systems. While in both contexts we have analysed we are looking at classifiers in a security environment (malicious web scraping detection and malicious software sample classification), the methodology we describe can be used to assess any binary classification systems.

3.1 Building classifiers

The initial step that underpins the assessment of any classifier system is the creation of said system. In this work (as is the case in many production scenarios), the building of classifiers is a separate endeavour, many times delegated to third party companies whose off-the-shelf products are simply licensed. We can nonetheless describe the building of classifiers, as this will allow for an added insight into their assessment, particular in our work looking at design diversity between created classifiers.

In our first body of work, we have looked at the detection of malicious web scraping activity. This analysis focused on the classifications generated by a set of two intrusion detection systems, CommTool (anonymised) and Arcane, tasked with analysing the HTTP traffic observed on a production network. In our analysis we treat both of these tools almost entirely as black boxes, noting only the most fundamental differences between them. CommTool is a commercial tool that works primarily on the client-side, sending JavaScript tests to clients, while Arcane, being an internally built tool by the company which provided us with the data - Amadeus - working exclusively server-side, looking at, and correlating between various HTTP requests and their meta-data. Because we are treating these as black boxes, when studying their performance in chapter 4, we mostly do so based on the differences between samples they classify and then try to correlate these to how the tools operate functionally.

Our second body of work includes a more detailed description of how the classifiers were built. We analyse the predictions generated by a set of 37 RNN machine learning models making classifications on the nature of software samples - identifying between benignware and malware. Each of these 37 models was built using a different set of hyperparameters values (e.g. number of hidden layers, choice of optimiser, learning rates, etc). As we will make clearer in the relevant chapter, the knowledge of how the RNNs were built, and how they internally differ from one another gives us a great insight into explaining the diversity inherent in their classifications, as well as significantly aiding in how the analysis is performed - by for example looking at differences between models using different or similar values of a particular hyperparameter of note.

More details on the construction and nature of both the malicious web scraping detectors and the RNN models used for classification of software samples can be found in their respective chapters 4 and 5.

Finally, in our last chapter of work we look at the use of optimal adjudication. As opposed to the previous two analysis, we do carry out the creation and validation of optimal adjudicators. However, because the creation of optimal adjudicators necessitates the collection of data from the single classifiers that constitute it, we discuss this in section 3.3.

3.2 Validating classifiers

When validating classifiers, we are interested in looking at how they perform on a representative workload. What this workload entails will vary based on the context in which we are working on. The workload that the classifiers are subjected to ought to be representative of what the classifiers will be expected to encounter in a production environment. Additionally, in order to validate between different classifiers, it is crucial that all of the samples in our workload are seen by all of the classifiers, such that no classifiers have access to different pieces of information.

Extra care should be taken when validating classifiers which require a training and validation approach, such as machine learning models. Before generating a classification for a particular sample, each classifier must have been trained on the same set of training data (unless of course our goal is to look at classifiers which have been trained independently from one another). This insures that the classifiers are directly comparable.

At this stage, and when dealing with binary classifiers, our only requirement is that we gather enough information to be able to create a confusion matrix for each individual classifier (Table 4). Thankfully, every performance metric we have previously outlined in section 2.2.1, and which we will use at a later point to assess between classifiers, can be calculated from the data included in a confusion matrix, thus the selection of which metrics to utilise is not necessary at this stage.

Table 4: Binary classification system - confusion matrix

	Actual: Positive	**Actual: Negative**
Classification: True	TP	TN
Classification: False	FP	FN

These confusion matrices can be generated for both individual classifiers, as well as combinations of classifiers. This is done by, instead of looking at the predictions generated by a single classifier, looking at the predictions from all of the classifiers in question and deciding, based on the adjudication scheme being used (e.g. 1ooN or NooN), what the overall system would have outputted.

In order to generate these confusion matrices from previously collected data, there are two requirements:

- The data must includes the predictions generated by each classifier for each sample;
- The data must includes the truthful classification of each sample;

Additional ”meta-data” about the nature of the samples analysed, or the classifiers making predictions, can allow for a more in-depth analysis. In our work with malicious web scraping detection, we did this in regards to the samples being analysed, regarding HTTP request meta-data such as number of bytes sent or received, HTTP error status code, etc. Similarly, in our work with machine learning models for software classification we not only looked at the types of malware samples present in the dataset, but also the various hyperparameters that differentiate between the various RNN models used.

3.3 Building optimal adjudicators

The creation of optimal adjudicators can only take place after collecting validation data for the classifiers we wish to use in our composition. As described in section 2.2.3, an optimal adjudicator is an adjudication function that attempts to map each unique combination of outputs from it's internal classifiers - known as a *syndrome* - into a single, overall output, based on the number of different types of samples which trigger that same syndrome. Each of these unique syndromes partition the sample space of our system into disjoint subsets, such that each sample triggers one, and only one syndrome. In order to build an optimal adjudicator then, we must look at the predictions generated by individual classifiers for the same set of classification samples.

We begin by defining all of the possible syndromes for a given combination of classifiers, which allows us to create a table like the one shown in Table 37, where for each syndrome we tally up the number of positive and negatives samples which generate that syndrome.

Table 5: Optimal adjudicator example

	Positives	**Negatives**	**Optimal adjudication**
A = 0 **B = 0** **C = 0**	139	1418	**No alert**
A = 1 **B = 0** **C = 0**	20	107	**No Alert**
A = 0 **B = 1** **C = 0**	420	28	**Alert**
A = 0 **B = 0** **C = 1**	20	18	**Alert**
A = 1 **B = 1** **C = 0**	87	272	**No alert**
A = 1 **B = 0** **C = 1**	15	11	**Alert**
A = 0 **B = 1** **C = 1**	112	17	**Alert**
A = 1 **B = 1** **C = 1**	1112	270	**Alert**

When dealing with binary classifiers, we are faced with one of two possibilities: our classifiers may be either *strict binary classifiers* - classifiers which only output one of two values: either 0 (negative) or 1 (positive) - or *non-strict binary classifiers* - classifiers that generate a value between two extremes (such as a confidence value between 0 and 1) which can then be mapped to either classification based on whether the generated value is above or below some predetermined "threshold" point between the two extremes.

For strict binary classifiers, the syndromes that make up an optimal adjudicator are just all of the possible combinations of outputs. For non-strict binary classifiers we require the use of threshold values in order to determine all of the possible syndromes. If using just one threshold, the situation is directly analogous to strict classifiers, as we can convert values below the threshold to 0, and values above the threshold to 1. We can however, expand the number of thresholds to allow us more granularity in our definition of syndromes. For this, we could define two thresholds for example, at points 0.33 and 0.66, allowing us to define syndromes such as ($A \leq 0.33$, $0.33 < B \leq 0.66$, $C \leq 0.33$).

After constructing a table such as the one above, we can determine the output of an optimal adjudicator on a syndrome by syndrome bases, simply by selecting the option (either 0 or 1) which generates the least amount of false classifications. In this sense, an optimal adjudicator, much like a machine learning model, requires a training and testing split, where a portion of the data is used to "train" each syndrome of the optimal adjudicator (i.e. determine what the output of that syndrome should be), and another portion is used to validate the actual performance of the optimal adjudicator, in order to capture the inaccuracies that may have been present in the training portion, and such that we avoid what are known as "overfitting" problems - situations where models are trained on such specific training data such that they essentially "recognise" individual samples rather than the underlying nature of what constitutes a positive or negative sample.

We delve deeper into the creation of optimal adjudicators, and expand on it's intricacies in chapter 6.

After generating an optimal adjudicator function, we can validate it as we would with any other classifier, by supplying it with a workload and create a confusion matrix as we describe in the previous section.

3.4 Metric selection

In section 2.3 we highlighted a large number of different metrics that can be used to assess the performance of binary classifiers. All of these metrics are calculated based on the number of true and false positives and negatives, found in the confusion matrices we generated in the previous step, and any number of them can be used to assess the performance of a classifier (be they individual classifiers or combinations of classifiers using conventional adjudication schemes or optimal adjudication schemes).

Not all metrics however, are relevant for all assessments, and the selection of which metrics to use is going to be dependent on the goal of the analysis and the context in which our classifiers operate in. Vieira et. al. provide a very detailed methodology on the selection of adequate metrics [3]. In it, the authors expand on the characteristics of an assessment metric, such as whether they are repeatable, consistent, comparable, understandable and meaningful. All of the metrics we have present are repeatable, consistent and comparable. The degree to which they are understandable varies, however this does not take away from the usefulness of the metrics. What is most important is the meaningfulness of the metrics, and how contextually advantageous they are. Some contexts, such as medical diagnosis, put a higher importance on not generating false negatives, and so, metrics which are calculated using these numbers are desirable, while in other contexts, such metrics might not be as useful.

In this work, we have only dealt with security environments, so both correctly identifying positives, and not generating falsely classifying benign samples as malicious are important when assessing the tools we have looked at (although the degree to which one is more important can vary). There are two primary metrics that we look at for this purpose - sensitivity and specificity. These two metrics, as explained before, are well known and often used to analyse intrusion detection tools, and as such, are easily comparable to other studies and analyses.

In our first body of work, looking at the detection of malicious web scraping activity, we make use of two additional metrics. Firstly, we use precision. In this context, not generating false positives takes prevalence over identifying malicious activity because blocking legitimate users has a more detrimental effect than not detecting any one instance of malicious activity. As such, precision allows us to understand, from the pool of total positives generated, which were correctly classified as positives. Another aspect of the datasets we analysed is that they are bot-heavy, meaning that there are a lot more positive samples than negative ones. Because of this, it is important that we use a metric that can adequately take into account differently-sized sets of classes, which is the case for Matthew's correlation, as explained in Section 2.3.5.

3.5 Analysing for design diversity

The steps we have talked about thus far in our methodology are useful for assessing the performance of any binary classifier system, and to be able to compare between different classifiers on a high level. Further to this, we are interested in understanding the design diversity that exists between the different classifiers we have looked at, and so we need to be able to correlate between different aspects of both the samples being classified, as well as the classifiers themselves and how they operate. Most importantly is understanding what makes classifiers have different classification patterns, and how they may be combined such that we can create the best performing overall system.

When dealing with black box classifiers, such as our work with malicious web scraping detectors, we are limited to understanding these differences in operation based solely on the types of samples being classified. The hope is to identify what makes samples easier for any particular classifier to correctly identify, and in our cases this comes down to the various meta-data information for each sample. Calculating performance metrics (sensitivity, specificity, precision and Matthew's correlation) should be done based on different subsections of samples, for example looking at the performance achieved by the classifiers when looking only at high byte count samples vs. low byte count. Doing this would highlight potential reasons that influence how the classifiers perform, drawing indications to situations where one classifier might supersede another and vice versa.

In a similar manner, when looking at combinations of classifiers, we can devise combinations that separate classifiers along their inherent properties. We do this extensively in our second body of work, looking at the classification of software samples using RNN machine learning models, where we deliberately try to compare combinations of classifiers that have different values of hyperparameters, for example, combinations where all individual classifiers have similar values of sequence length vs. combinations where the individual classifiers have different values of sequence length. When dealing with a reasonable number of classifiers, as is the case in work, applying these techniques leads to potentially identifying the more important properties to creating diverse classifiers. This knowledge can afterwards be used in later iterations to create the most diverse classifiers possible, such that when combined using diversity diversity we can achieve a superior overall system.

Where possible, all such properties of samples and classifiers should be analysed in this manner, and more still, combinations of properties can be highlighted to understand potential interactions between properties and how different values can lead to more or less diverse classifiers. For cases where this is not possible, some context specific determination of what makes an "important" property would be necessary, and this will depend on the nature of the classifiers being used and the context in which they integrate.

3.6 Conclusions

In this chapter we have detailed our methodology for performing the analysis we will present in this thesis, including the creation of both single classifiers and optimal adjudicators, validation of classifiers and the selection of metrics for assessing classifier performance. We provide more detailed methodology descriptions at the start of each subsequent chapter which pertain to the specific studies we have undertaken.

4 Diverse web scrapers

In this chapter we present our analysis of three datasets, consisting of HTTP traffic flows collected from live environments, which are operated by our partner Amadeus, a multinational IT provider for the global travel and tourism industry. The datasets contain the traffic flows observed on their network, as well as the alerts generated by two intrusion detection tools set to defend against malicious web scraping activity. This work was started as part of the EU funded DiSIEM project [11], in which City, University of London and Amadeus were part of the consortium.

We have analysed the three datasets individually as they were provided to us, and in this chapter we will present the results highlighting two major differences between how the datasets were collected. The first and second dataset were collected on the same domain, but at different times, and so the comparison between these two datasets will focus on how the web scraping detectors improved over time. The second and third datasets were collected at the same time but from different domains, and so the comparison between these two datasets will focus on the ability of the tools for tackling different environments.

This chapter is outlined as follows: in section 4.1 we describe the three datasets we have analysed and what their differences entail for the results we will present. Section 4.2 describes in more detail the two web scraping detectors which are present in all three datasets, and how they differentiate from one another in the way they operate. Finally in section 4.3 we present the results of our analysis. We first present overview results, moving on to the diversity aspects which we observed, and finally describe the various features which are causing the web scraping detectors to diverge in their alerting patterns; finally in 4.4 we discuss the conclusions of these analysis. An application of optimal adjudication analysis on these datasets is presented separately, in Section 6.

4.1 Description of the datasets

All three datasets we have analysed consist of Apache HTTP access logs gathered from two similar e-commerce applications. For the remainder of this report, we will refer to these datasets as *D1*, *D2* and *D3*.

Each demand in the datasets represents an HTTP request session and contains HTTP request data (useragent, method, etc.), HTTP response data (status, content-type, etc.), as well as metadata about the connection, such as duration and bytes sent and received. Each demand in the datasets was analysed by two intrusion detection tools at the time of the capture, named *Arcane* (in-house tool built by Amadeus) and *CommTool* (this is the anonymised[10] name of a commercial tool utilised by Amadeus in their environment), and subsequently classified into either "malicious bot traffic" or "non-bot traffic". These tools are further explained in the next section (4.2).

Alongside the classifications generated by the web scraping detectors, each demand present in the datasets was also manually classified as either "malicious bot traffic" or "non-bot traffic" by engineers at Amadeus. This manual classification effort is based on their experience with detecting malicious web scraping activity, and allows us to understand and pinpoint which classifications made by the web scraping detectors were correct

[10]Anonymised at the request of the dataset provider.

or incorrect, and where the tools disagreed with one another.

Dataset D1 was the first dataset to be collected and analysed, and covers a period of 5 days, from May 7th to May 12th, 2018. After we completed our analysis of D1, the results we obtained were used by the dataset provider to tweak both Arcane and CommTool. A new dataset was then captured, D2, using the tweaked web scraping detectors, which spanned a period of 6 days, from September 13th to September 19th, 2018. Both datasets come from the same domain, which is geographically located in Europe.

The third dataset, D3, was captured during the same time period as D2, from September 13th to September 19th, 2018, but originates from a different domain, which was not subject to the same traffic and attacks, and where the web scraping detectors (Arcane and CommTool) were not tweaked in accordance with the results from the analysis done on D1. As opposed to D1 and D2, dataset D3 was captured from a domain located in South Asia.

4.2 Description of the detection tools

In this subsection we provide additional details about the two web scraping detectors present in all three datasets we have analysed.

4.2.1 CommTool

CommTool, whose name is anonymised in this report, is a commercial tool used to detect and manage bot activity on websites. The dataset provider uses a version of CommTool deployed in the cloud in front of the web servers of the web applications they protect. This means that all HTTP requests coming from users are first inspected by CommTool before being redirected to the application servers. Legitimate requests are forwarded to the web application and requests deemed as originating from malicious bots are blocked. CommTool (as well as many other web scraping detectors) uses different techniques to detect malicious bots:

- Client-side fingerprinting: A JavaScript file is downloaded from the protected website and run on the client's browser. This script extracts many device attributes to create an accurate fingerprint of the client's system. The fingerprints are shared worldwide amongst CommTool products, creating a global database of known violator fingerprints.

- JavaScript tests: For suspicious use sessions, CommTool can run further client-side JavaScript tests, such as inspecting the consistency of device attributes.

- Machine learning: CommTool uses evolving behavioural user models based on the data collected from different domains protected.

- Custom rules: For advanced scraping activities that are not detected using the above methods, CommTool can implement custom rules based on user device attributes or specific HTTP headers. Custom rules can also be created on request by the customers of the tool (e.g., to monitor a particular domain of interest to the customer).

- Known violator databases: CommTool uses a worldwide database of known violators for easy identification of bots. The known violators can be User-Agents, Browser fingerprints, IP addresses, subnets, ISPs, organisations, or countries.

4.2.2 Arcane

Arcane is an internal tool, built in-house by the dataset provider, and used to detect web scraping activity. Arcane is used to monitor CommTool's performance for domains which are already protected by CommTool, and assess robotic activities for non-protected domains. Unlike CommTool, Arcane is a purely server-side tool, meaning that it works on, and only has access to, information seen by the server when it receives a request. It uses Apache HTTP access logs and the information these contain to detect scraping activity, with HTTP access records being grouped into HTTP sessions via a unique session identifier. The unique session identifier is stored in the client's browser cookies and logged in the Apache audit trails. From these sessions, Arcane collects a variety of different features, which it uses to determine whether a request was originated by a malicious web scraping bot, or by a human. The different features used by Arcane are shown in Table 6.

Table 6: Arcane session features

Feature	**Description**
Session Duration	The duration of a session.
Number of requests per session	The number of requests per sesion.
Burst rate	The maximum number of requests made during an adjustable sliding time window.
Static/Dynamic request ratio	The ratio between the number of requests made to static resources and the number of requests made to dynamic pages.
Entropy	A measure of the diversity of URL paths requested.
Number of IP addresses	The number of distinct IP addresses used during the same session.
Number of user agents	The number of distinct user agent values in a session.
Number of HTTP errors	The number of requests that results in 4xx and 5xx error pages during a session.
Ratio of "availability" requests	The ratio between the number of requests made to sensitive pages (i.e., content-rich pages such as fares, availability of seats, prices) and the total number of dynamic requests during a session.

For each of the features in the table, a manually defined scoring function is used to compute an anomaly score. These scores are then aggregated into a global anomaly score used to decide whether an HTTP session originated from a web scraping bot or from a benign source. The dataset provider uses a custom known violator database, similar to that of CommTool. While the dataset provider uses CommTool in blocking mode, every HTTP request is still observed by both web scraping detectors, with Arcane analysing the traffic after the fact.

4.3 Results

4.3.1 Overview

In Table 7, we show the total number of requests present in all three datasets, as well as what proportion of these were determined to be malicious bot traffic by the dataset provider. In all three datasets, we are dealing with attack-heavy data, with D1 having 96%, D2 having 98% and D3 having 83% malicious bot traffic. The travel industry is particularly targeted by scraping activity, because of the type of content that can be scraped (fares and availability). Although the proportion of bot traffic in our datasets is high, the dataset provider reassured us that this proportion is not uncommon across the different domains that they monitor.

Tables 8, 9 and 10 show how both Arcane and CommTool classified HTTP requests that were manually determined to be bot traffic, in D1, D2 and D3, respectively. Numbers under *reported* identify the true positives generated, while *not reported* identify false negatives generated by each tool. The same information is shown in Tables 11, 12 and 13, for HTTP requests that were manually determined as non-bot traffic, where *reported* identifies false positives, and *not reported* identifies true negatives. We can see that, with a few exceptions, both web scraping detectors tend to overlap on the true positives or true negatives across the datasets.

Table 7: HTTP request counts for D1, D2 and D3

	D1	**D2**	**D3**
Total requests	5,028,429 (100%)	4,349,618 (100%)	10,522,982 (100%)
Total scraping requests	4,825,011 (96%)	4,250,418 (98%)	8,723,877 (83%)
Total non-scraping requests	203,418 (4%)	99,200 (2%)	1,799,105 (17%)

Table 8: Scraping Requests Analysed by Both Tools for D1

Web scraping requests		**Arcane**		**Totals**
		Reported	**Not reported**	
CommTool	**Reported**	1,369,281	146,998	1,516,279
	Not reported	1,430,200	1,878,532	3,308,732
Totals		2,799,481	2,025,530	4,825,011

Table 9: Scraping Requests Analysed by Both Tools for D2

Web scraping requests		**Arcane**		**Totals**
		Reported	**Not reported**	
CommTool	**Reported**	4,060,181	28,937	4,089,118
	Not reported	135,568	25,732	161,300
Totals		4,195,749	54,669	4,250,418

Table 10: Scraping Requests Analysed by Both Tools for the D3

Web scraping requests		**Arcane**		**Totals**
		Reported	**Not reported**	
CommTool	**Reported**	3,038,246	666,218	3,704,464
	Not reported	4,496,094	523,319	5,019,413
Totals		7,534,340	1,189,537	8,723,877

Table 11: Non-Scraping Requests Analysed by Both Tools for D1

Non-scraping requests		**Arcane**		**Totals**
		Reported	**Not reported**	
CommTool	**Reported**	353	270	623
	Not reported	43,356	159,439	202,795
Totals		43,709	159,709	203,418

Table 12: Non-Scraping Requests Analysed by Both Tools for D2

Non-scraping requests		**Arcane**		**Totals**
		Reported	**Not reported**	
CommTool	**Reported**	4,132	1,910	6,042
	Not reported	1,645	91,513	93,158
Totals		5,777	93,423	99,200

Table 13: Non-Scraping Requests Analysed by Both Tools for the D3

Non-scraping requests		**Arcane**		**Totals**
		Reported	**Not reported**	
CommTool	**Reported**	3,391	6,199	9,590
	Not reported	15,357	1,774,158	1,789,515
Totals		18,748	1,780,357	1,799,105

From these confusion matrices we can calculate the performance metrics we first identified in section 2.3. Every metric we introduced was calculated, and of those we've selected four which fit with the purpose of our analysis and context of malicious web scraping detection. The first two of these are sensitivity and specificity, which are widely used in security contexts to account for the number of benign or malicious samples which were correctly identified, respectively. Sensitivity and specificity are calculated using the following two formulas, and the values achieved by both Arcane and CommTool are shown in Table 14.

$$Sensitivity = \frac{TP}{P} \tag{13}$$

$$Specificity = \frac{TN}{N} \tag{14}$$

What we can observe is that, for D1, Arcane presents itself as a more sensitive, but less specific system, when compared to CommTool. For D2, the metrics of each system are more homogeneous. With the exception of CommTool's specificity, which decreased when compared to its rate in D1, all metrics have risen above 94%. From a very high-level perspective, this suggests that both Arcane and CommTool were diverse in their alerting behaviours for D1, and that, after being tweaked, the diversity between them diminished as each tool individually became more reliable. Looking at the comparison between D2 and D3, D2 has a better sensitivity, in terms of both tools, while D3 experiences a higher specificity rate for both Arcane and CommTool.

Table 14: Arcane and CommTool Sensitivity and Specificity for D1, D2 and D3

	D1		**D2**		**D3**	
	Arcane	**CommTool**	**Arcane**	**CommTool**	**Arcane**	**CommTool**
Sensitivity	58.02%	31.42%	98.71%	96.21%	86.36%	42.46%
Specificity	78.51%	99.69%	94.18%	93.91%	98.96%	99.47%

While sensitivity and specificity are often the two most used metrics to measure the performance of security tools, as we highlighted in our literature review section, they are not the only ones, and are not necessarily the most effective. Precision is an often-used metric, which gives an assessor an idea of how many of the positives generated by a tool were correctly generated (i.e. how many of the alerts we generated were actually for malicious samples). The higher the value of precision, the less false positives were generated by a tool, and this insight can be useful if generating a false positive is particularly costly. This is the case for these datasets, where preventing a legitimate user from interacting with the store front will directly prevent that user from making a purchase, while missing a malicious actor might have a very small impact in the amount of damage it can do (in the context of price scraping). The value of precision is calculated using the formula:

$$Precision = \frac{TP}{TP + FP} \tag{15}$$

Another aspect that we need to account for in these datasets is that they are bot-heavy, meaning that most of the samples are from malicious actors and very few from legitimate users. In other words, we need to take into account this disparity between a large number of positive samples and a very small number of negative samples when measuring the performance of the tools. The use of Matthew's correlation can account for this disparity in sample types, in a sense measuring both the sensitivity and specificity of a tool into a single value for easier comparison. Matthew's correlation is calculated using the following formula:

$$Matthew's\ correlation = \frac{TP * TN - FP * FN}{\sqrt{(TP + FP) * (TP + FN) * (TN + FP) * (TN + FN)}} \tag{16}$$

In Table 15, we present the values of Matthew's Correlation (MC) and Precision for both Arcane and CommTool, for all three datasets. In it, we see that Matthews Correlation shows a big increase from D1 to D2, for both Arcane and CommTool. This jump in performance was observable when analysing sensitivity and specificity, although Arcane's increase is more pronounced using this metric. At the same time, precision does not change much from D1 to D2, meaning that the proportion of users being incorrectly blocked is roughly the same. We see a different story looking at the metrics comparing D2 and D3 - Arcane is very comparable between the two domains, while CommTool has a slight difference in terms of its Matthew's Correlation value.

Table 15: Arcane and CommTool Matthew's Correlation and Precision for D1 and D2

	D1		**D2**		**D3**	
	Arcane	**CommTool**	**Arcane**	**CommTool**	**Arcane**	**CommTool**
MC	0.1452	0.1336	0.7647	0.5732	0.7137	0.3303
Precision	98.46%	99.96%	99.86%	99.85%	99.75%	99.74%

4.3.2 Diversity analysis

In monitoring environments, adjudication systems are often used to decide whether observed items are malicious or not based on the decisions of multiple monitors. Based on the counts in overview tables we presented previously, we can extrapolate the results that we would observe under 1oo2 and 2oo2 adjudication systems, in order to understand what kind of improvements, or deterioration, can be harnessed from using both web scraping detectors together. Tables 16 and 17 show how both of these adjudication schemes would have analysed malicious bot and non-bot HTTP requests, respectively. We extend this in Table 18, where we show the sensitivity and specificity rates for both adjudication systems.

Table 16: Web Scraping Requests Analysed by Adjudication Systems for D1, D2 and D3

	D1		D2		D3	
	Reported	**Unreported**	**Reported**	**Unreported**	**Reported**	**Unreported**
1oo2	2,946,479	1,878,532	4,224,686	25,732	8,200,558	523,319
2oo2	1,369,281	3,455,730	4,060,181	190,237	3,038,246	5,685,631

Table 17: Non-Scraping Requests Analysed by Adjudication Systems for D1, D2 and D3

	D1		D2		D2	
	Reported	**Unreported**	**Reported**	**Unreported**	**Reported**	**Unreported**
1oo2	43,979	159,439	7,687	91,513	24,947	1,774,158
2oo2	353	203,065	4,132	95,068	3,391	1,795,714

Table 18: Adjudication Systems Sensitivity and Specificity for D1, D2 and D3

	D1		D2		D3	
	1oo2	**2oo2**	**1oo2**	**2oo2**	**1oo2**	**2oo2**
Sensitivity	61.07%	28.38%	99.39%	95.52%	94.00%	34.83%
Specificity	78.37%	99.83%	92.25%	95.83%	98.61%	99.81%

Changing from any system individually to a 1oo2 adjudication scheme increases sensitivity, while lowering specificity. The inverse occurs when changing any individual system to a 2oo2 adjudication scheme. Both of the changes are expected, as explained before when we described adjudication systems (section 2.2.2). What is important to take from this is how much better, or how much worse, a diverse pair would perform in these setups, and the results in Table 18 give us some indicators about this. It would be up to the corresponding security administrators, and dependent on the system context, on which improvements would be most beneficial or cost-effective.

Again, as with the individual tools, we present in Table 19 the values of Matthew's Correlation (MC) and Precision calculated for adjudication systems. This time, the comparisons that can be drawn are the same as with the individual tools, although they are more pronounced when looking at adjudication systems.

Table 19: Adjudication systems' Matthew's Correlation and Precision values for D1, D2 and D3

	D1		D2		D3	
	1oo2	**2oo2**	**1oo2**	**2oo2**	**1oo2**	**2oo2**
MC	0.1583	0.1248	0.8448	0.5509	0.8440	0.2877
Precision	98.53%	99.97%	99.82%	99.90%	99.70%	99.89%

4.3.3 Diversity based on CommTool flags

So far, we have demonstrated the presence of diversity in the alerting patterns of both tools, but we have not said anything about the source of this diversity, i.e., where the diversity is coming from. The way in which both Arcane and CommTool detect malicious web scraping activity, and the different techniques they employ is the main reason for this difference. Tables 20 and 21 show the number of CommTool and CommTool-only true positives based on the reasons the tool flagged the connections for, comparing D1 and D2. Tables 22 and 23 show the same information but comparing D2 with D3. In some cases, multiple reasons were used by CommTool to flag a request, and in these cases, we have counted the alert in more than one category. These flags given by CommTool indicating the reason for the alert is a characteristic only found in CommTool, hence why we have not carried out the same analysis regarding Arcane.

Table 20: CommTool Reasons for True Positives for D1 and D2 (Shortened)

D1		**D2**	
Reason	**Count**	**Reason**	**Count**
JavaScript Check Failed	1,422,528	**Known Violator User Agent**	7,969,663
JavaScript Not Loaded	797,420	**Pages Per Session Exceeded**	101,824
JavaScript Not Loaded & JavaScript Check Failed	728,785	**JavaScript Check Failed**	54,988
Rate Limited	671,081	**JavaScript Not Loaded**	19,291
Known Violator Data Centre	157,920	**Known Violator Data Centre**	19,248
Pages Per Session Exceeded	36,531	**JavaScript Not Loaded & JavaScript Check Failed**	11,086
Session Length Exceeded	19,664	**Known Violator**	11,591

Table 21: CommTool-Only Reasons for True Positives for D1 and D2 (Shortened)

D1		**D2**	
Reason	**Count**	**Reason**	**Count**
JavaScript Check Failed	91,408	**Known Violator User Agent**	45,954
JavaScript Not Loaded	90,741	**Known Violator**	3,958
JavaScript Not Loaded & JavaScript Check Failed	90,090	**Pages Per Session Exceeded**	2,961
Rate Limited	77,567	**Known Violator Data Centre**	2,947
Known Violator Data Centre	26,750	**IP Pinning Failure**	745
Known Violator	5,866	**JavaScript Check Failed**	557
IP Pinning Failure	1,299	**JavaScript Not Loaded**	541

As explained before, CommTool is primarily a client-side tool that works by creating a fingerprint of the client's device and matching this fingerprint to a known violator database. This fingerprint is generated early on in a session, through the use of JavaScript. As such, JavaScript checks are the most common reason for CommTool's true positives for D1, and the third most common for D2. Since Arcane does not utilise JavaScript in its detection procedure, this is theoretically a significant source of diversity between the tools. Nevertheless, Arcane is still able to detect most of the same connections detected by CommTool's JavaScript process (roughly 93.5% for D1 and 99% for D2) since these connections are initiated by hosts that are in Arcane's known violator database, for reasons other than JavaScript violations. The reasons for CommTool alerts are roughly the same between all CommTool true positives and CommTool-only true positives for D1, with the exception of *Pages Per Session Exceeded.* This is because Pages Per Session is also a measure found in Arcane, and as such it does not provide a meaningful source of diversity between the tools. For D2 we see that JavaScript-related reasons are less common in CommTool-only alerts, while Known Violator reasons (Known violator IPs, subnets, etc) rise. This implies that the diversity between Arcane and CommTool during D2 is coming from the fact that the tools are using separate known violator databases, rather than any inherent diversity between their operating methodology.

Table 22: CommTool Reasons for True Positives (Shortened) for D2 and D3

First Domain		**Second Domain**	
Reason	**Count**	**Reason**	**Count**
Known Violator User Agent	7,969,663	**Pages Per Session Exceeded**	1,922,660
Pages Per Session Exceeded	101,824	**Known Violator**	1,338,137
JavaScript Check Failed	54,988	**JavaScript Check Failed**	1,155,995
JavaScript Not Loaded	19,291	**JavaScript Not Loaded**	1,128,554
Known Violator Data Centre	19,248	**JavaScript Not Loaded & JavaScript Check Failed**	927,247
JavaScript Not Loaded & JavaScript Check Failed	11,086	**Unique Identifier ACL**	890,940
Known Violator	11,591	**Known Violator & Pages Per Session Exceeded**	627,836

Table 23: CommTool-Only Reasons for True Positives (Shortened) for D2 and D3

First Domain		**Second Domain**	
Reason	**Count**	**Reason**	**Count**
Known Violator User Agent	45,954	**Pages Per Session Exceeded**	727,492
Known Violator	3,958	**Known Violator**	382,859
Pages Per Session Exceeded	2,961	**Known Violator & Pages Per Session Exceeded**	231,898
Known Violator Data Centre	2,947	**IP Pinning Failure**	94,704
IP Pinning Failure	745	**JavaScript Check Failed**	57,348
JavaScript Check Failed	557	**Pages Per Session Exceeded & IP Pinning Failure**	48,195
JavaScript Not Loaded	541	**JavaScript Not Loaded**	38,945

When comparing the different domains in D2 and D3, we see that for D2 the vast majority of CommTool's alerts are generated based on Known Violator information, while the alerts generated in D3 are more evenly spread out between Known Violator information, JavaScript errors and user behaviour. This could help to explain CommTool's sensitivity decrease in D3 when compared to D2, as this likely indicates that a large proportion of the attacks made to D3's domains originated from IPs not present in Known Violator databases.

4.3.4 Diversity based on bytes sent

The HTTP access logs in the datasets contain information about the number of bytes sent in the responses to the client. This allows us to compare both tools' alerts to requests, based on the number of bytes in the responses for these requests. Fig. 7, 8 and 9 show the number of true positives generated by Arcane, Arcane-only, CommTool and CommTool-only based on the number of bytes the HTTP responses had, for D1, D2 and D3, respectively.

In both D1 and D2 we can identify three clusters of activity, and these roughly line up with each other. The first cluster of activity is between 0 and 8,000 bytes for D1, and 0 and 12,000 bytes for D2. For this first cluster we see both tools active, with Arcane generally having more unique true positives than CommTool. This trend only inverts during the 4,000 to 6,000 bytes range for D1, where CommTool has a large number of unique true positives. The second cluster of activity concerns the range of 18,000 to 20,000 bytes for D1, and 16,000 and 22,000 bytes for D2. For both datasets, this activity range is dominated by CommTool, being the only tool that generates unique true positives. Finally, the third cluster of activity comprises between 30,000 and 36,000 bytes for D1, and 28,000 to 54,000 bytes for D2. In both datasets, this range of activity is mostly limited to Arcane alerts, with the exception of a reasonable number of CommTool alerts in D2, in the range of 34,000 to 36,000 bytes.

Looking at the results from D3, while these clusters are not as well defined, we still see that the activity spikes around the same areas. In Fig. 9 we see that there is increased activity between 0 and 8,000 bytes - similar to D2 domain's cluster - and an increased activity after the 32,000 bytes mark. We also see an outlier in activity between 20,000 and 24,000 bytes which is close to D2's second cluster of activity.

If we take a look at the same information for false positives, as shown in Figs. 10, 11 and 12, we can see that CommTool does have activity past the 22,000 bytes mark, however, it only generates false positives. After discussions with the dataset provider we were told that neither CommTool nor Arcane make use of the number of bytes sent for a particular request. Nevertheless, this appears as a source of additional observed diversity between the web scraping detectors. Because Arcane's output is based solely on HTTP access logs, its detection tends to improve as the duration of sessions increase. Over time, Arcane has access to more data and statistics that allow for a greater confidence in the alerts generated. This could help to explain why Arcane generates a higher number of alerts (both true and false positives) when compared to CommTool, as the number of bytes sent increases.

Taking a deeper look at the number of bytes sent in the requests of D1 and D2, we show in Figs. 13 and 14 the average number of bytes sent for true positives over time, for D1 and D2, respectively. This information is shown for true positives generated by Arcane, Arcane-only, CommTool and CommTool-only. What we see is that, for D1, both tools' average number of bytes sent is consistently higher for true positives they have uniquely produced when compared to all of their true positives (the ones they generated alerts for along with the other tool). This indicates that the alerting patterns of Arcane and CommTool appear to overlap for requests which generate responses with lower bytes, and as the number of bytes sent increases, they diverge in their alerting behaviour. This changes for D2, where Arcane and CommTool's lines appear almost identical, while the Arcane-only and CommTool-only lines diverge. This is consistent with previous observations that the tools have more alerts in common.

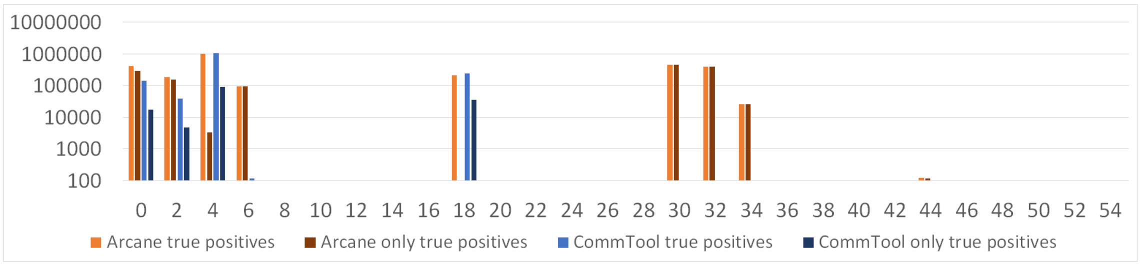


Figure 7: True positives by bytes sent for D1.

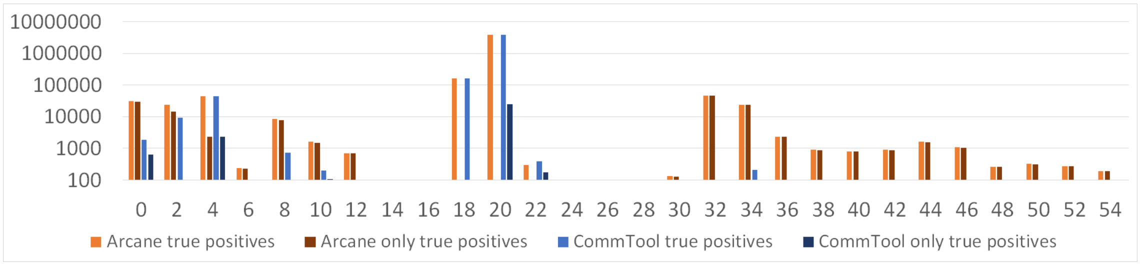


Figure 8: True positives by bytes sent for D2.

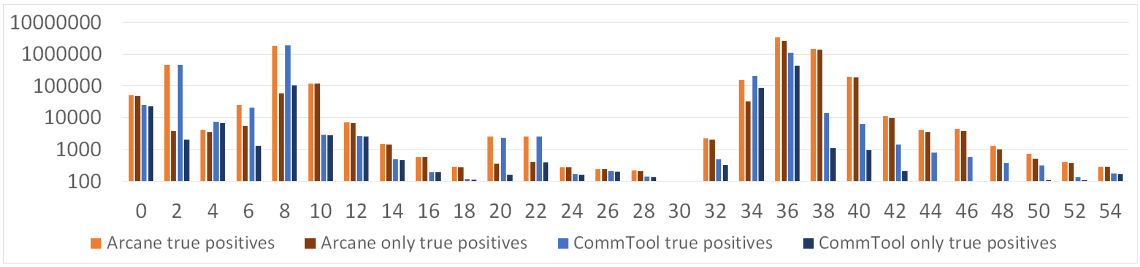


Figure 9: True positives by bytes sent for D3

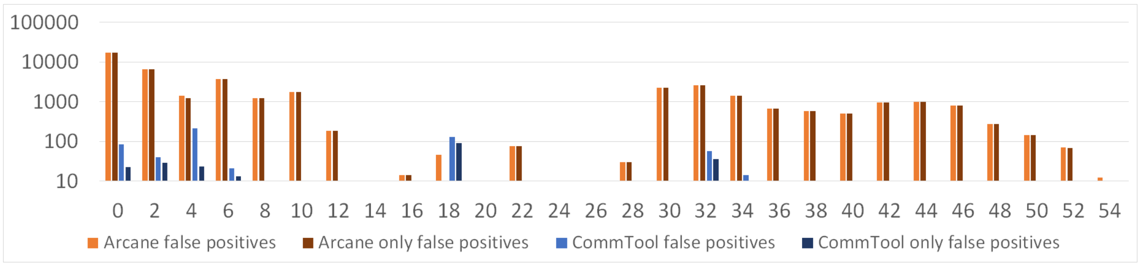


Figure 10: False positives by bytes sent for D1.

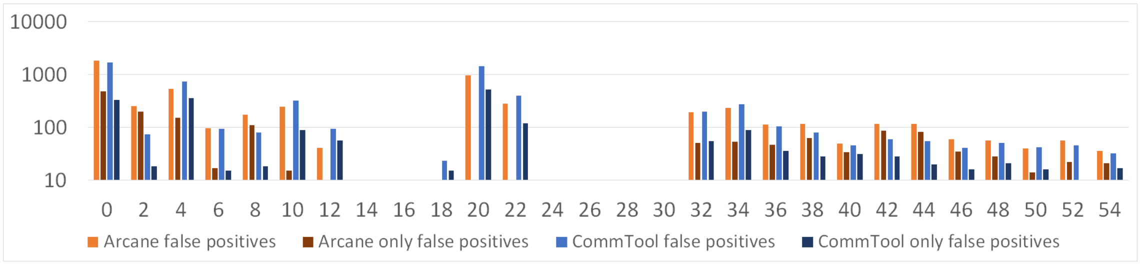


Figure 11: False positives by bytes sent for D2.

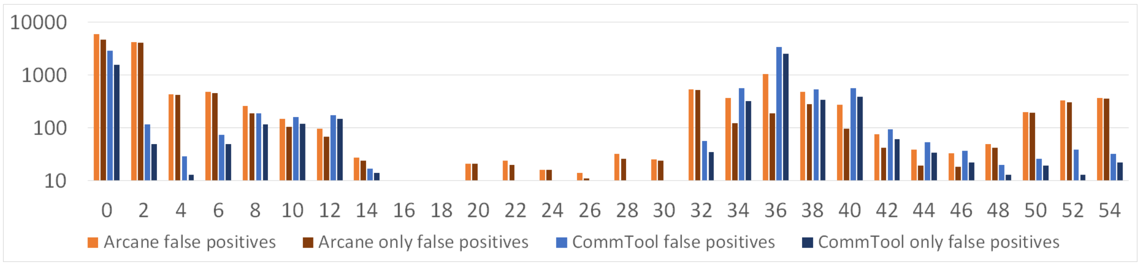


Figure 12: False positives by bytes sent for D3

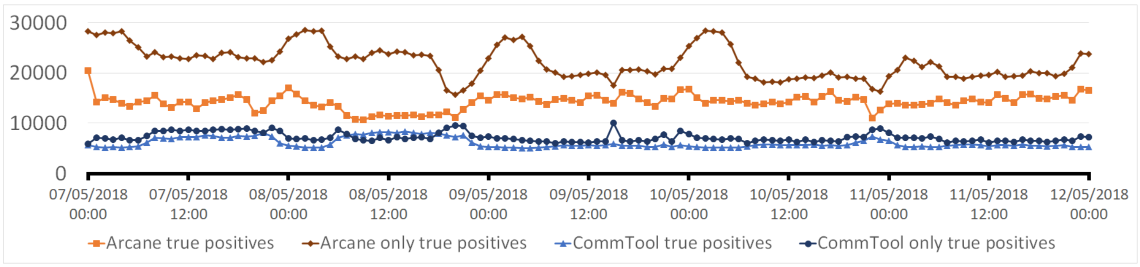


Figure 13: Average bytes sent for true positives for D1.

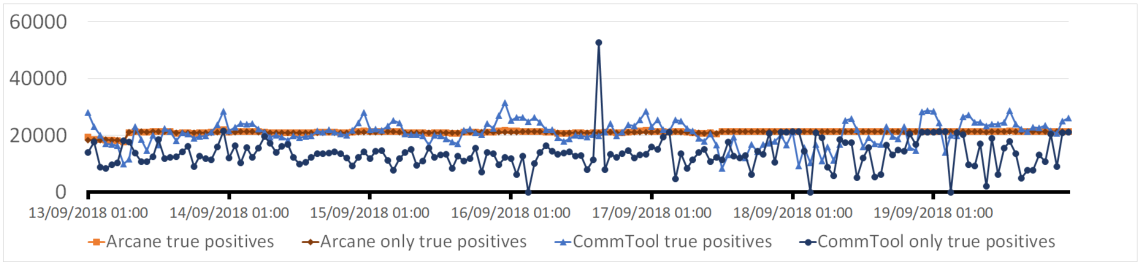


Figure 14: Average bytes sent for true positives for D2.

4.3.5 Diversity based on HTTP response status

Another area in which Arcane and CommTool seem to differ in terms of their alerting patterns is on the response status for HTTP requests. Tables 24, 25 and 26 are shortened versions of the true positive counts generated by Arcane, Arcane-only, CommTool and CommTool-only based on HTTP response status, for datasets D1, D2 and D3, respectively. The four HTTP status codes present in the tables are: 200 (OK), 302 (Found), 405 (Method not allowed) and 456, which is not an HTTP standard response status.

From Table 24 we see that response statuses 302 and 405 are alerted differently by Arcane and CommTool in D1. Investigating further, we could see that Arcane has a much higher number of unique alerts for HTTP response status 302 (Found), while CommTool is the only tool that generates unique alerts for status 405 (Method not found). Arcane does alert for requests whose response status was 405, however it does not do so uniquely, meaning that, for the case of HTTP response status 405, CommTool is presented as a superset of Arcane.

These observations hold true for D2 as well, where we can still see that CommTool is the only tool to generate unique alerts for HTTP status 405. From discussions with the dataset provider they confirmed that this is because CommTool sent a *CAPTCHA* at the start of the connection (with status 405). Because it was at the start of the connection (i.e., the CAPTCHA test was failed by the bot), CommTool detects it while Arcane does not. They also confirmed that most 302 requests are for "/". The HTTP sessions with 302 responses rarely contain requests to other pages on the website. Arcane was able to detect them because of its known violator database. CommTool did not detect them because the session origins are not in its known violator database, and the sessions were too short for CommTool to be able to run further JavaScript tests.

When comparing the different domains between D2 and D3, in general, the percentage of alerts generated by Arcane and CommTool are similar across the two datasets, with the only difference being that of status 456. Due to the nature of status 456 not being a standard HTTP response status, we are unable to provide further insight into this difference.

The similarities between the two datasets are more easily identified when we look at Arcane-only true positives for status 405, which happens to be 0 (zero). From discussions with the dataset provider they confirmed that this is because CommTool sent a *CAPTCHA* at the start of the connection (with status 405). Because it was at the start of the connection (i.e., the CAPTCHA test was failed by the bot), CommTool detects it while Arcane does not.

Table 24: True Positive Counts by HTTP Response Status for D1 (Shortened)

Status	Arcane TPs	CommTool TPs	Arcane only TPs	CommTool only TPs
200	2,167,085	1,123,320	1,138,231	94,466
302	418,115	143,578	291,710	17,173
405	213,929	248,825	0	34,896

Table 25: True Positive Counts by HTTP Response Status for D2 (Shortened)

Status	Arcane TPs	CommTool TPs	Arcane only TPs	CommTool only TPs
200	158,993	55,292	107,139	3,438
302	27,740	27	27,730	17
405	4,008,238	4,033,591	0	25,353

Table 26: True Positive Counts by HTTP Response Status for D3 (Shortened)

Status	Arcane TPs	CommTool TPs	Arcane only TPs	CommTool only TPs
200	6,246,804	1,943,729	4,456,126	153,051
405	801,216	1,304,705	0	503,489
456	445,639	446,274	0	635

4.3.6 Diversity based on user agent

One way to detect the presence of web scraping activities is to find evidence of botnets. This is because, attackers often lack the resources to establish a centralised scraping system, and resort to the use of botnets to launch their scraping attacks. Evidence of botnet activity can be found by looking at the parameters sent with each HTTP request, for example, looking at the value of the user-agent field. Under normal circumstances, where a human performs a request, this value takes on the name or description of the browser that the user utilised. By looking at the value of user agent across multiple unique requests generated by different client IPs, we can correlate them in order to detect the presence of botnets. Tables 27, 28 and 29 present the top user agents based on the number of total attack requests made (the user agents code meanings are presented in Table 30). For each of them we show how many requests originate from malicious bots, the number of unique IP addresses that have used this user-agent for malicious bot requests, and the number of alerts generated by Arcane, Arcane-only, CommTool and CommTool-only.

For D1 we see that Arcane has a large number of unique alerts generated for the top 4 user agents (which have been used by more than 1,000 unique IP addresses). After these first 4, Arcane ceases to alert uniquely for any user agent. Meanwhile, CommTool has a lower number of unique alerts for the top 4 user agents, when compared to Arcane, but from there on appears as a superset of Arcane (ie.e it detects all that Arcane does and more). These results are aligned with the way both tools work. Arcane is primarily server-side, therefore it has the ability to look at multiple HTTP requests and correlate them together to find botnets. As such, it is not surprising that it performs better for user agents which have been used by more unique IP addresses, as these potentially constitute evidence of larger botnets. What is surprising is that this behaviour changes for D2. In

D2, we no longer see this distinction where Arcane alerts uniquely for user-agents being used by more than 1,000 unique IPs and then stops alerting uniquely. For D2, both Arcane and CommTool correctly detect all attacks made by the top user agent, and from there onward they alternate in regard to which tool generates the most unique alerts.

For the results from D3, the number of Arcane-only true positives and CommTool-only true positives are not close between the two tools, with each useragent instance usually having one dominant tool, which generates the most unique alerts. This suggests that the tools have a much bigger overlap in terms of alerting patterns for D3.

Table 27: True Positives by Useragent for D1 (Shortened)

User agent	**Total attacks**	**Unique IP addresses**	**Arcane true positives**	**CommTool true positives**	**Arcane only true positives**	**CommTool only true positives**
(A)	2,775,396	162,149	983,747	537,247	540,126	93,626
(B)	1,109,404	21,195	1,037,946	469,769	581,221	13,044
(C)	705,335	14,783	550,182	293,541	292,321	35,680
(D)	230,682	1,053	226,742	211,528	16,532	1,318
(E)	789	297	590	789	0	199
(F)	610	313	23	610	0	587
(G)	559	271	11	559	0	548
(H)	505	269	16	505	0	489
(I)	169	42	34	169	0	135
(J)	56	28	0	56	0	56

Table 28: True Positives by Useragent for D2 (Shortened)

User agent	**Total attacks**	**Unique IP addresses**	**Arcane true positives**	**Comm-Tool true positives**	**Arcane only true positives**	**Comm-Tool only true positives**
(K)	3,960,416	380,221	3,960,416	3,960,416	0	0
(B)	173,794	16,769	167,063	88,893	84,277	6,107
(C)	38,358	7,728	38,306	10,783	27,575	52
(L)	6,568	2,959	2,025	613	1,647	235
(M)	1,537	2,153	1,297	533	1,004	240
(H)	15,324	1,767	7,305	7,732	6,006	6,433
(F)	13,769	1,718	3,782	7,394	2,945	6,557
(G)	10,338	1,674	3,498	7,605	2,733	6,840
(N)	2,322	1,215	635	196	512	73
(O)	704	831	660	176	528	44

Table 29: True Positives by Useragent for D3 (Shortened)

User agent	**Total attacks**	**Unique IP addresses**	**Arcane true positives**	**Comm-Tool true positives**	**Arcane only true positives**	**Comm-Tool only true positives**
(P)	5,834,459	538,186	5,834,459	1,662,397	4,172,062	0
(Q)	1,884,601	65,077	941,212	1,492,912	61,219	612,919
(L)	72,488	34,259	8,704	2,132	8,363	1791
(R)	15,317	31,782	8,458	7,086	8,231	6,859
(S)	78,875	24,372	78,875	33,667	45,208	0
(T)	52,179	19,453	51,799	32,973	19,206	380
(U)	6,948	19,443	2,238	4,830	2,118	4,710
(C)	46,123	17,047	45,661	29,835	16,288	462
(N)	30,096	14,446	3,766	1,167	3,447	848
(V)	104,807	10,987	104,807	69,060	35,747	0

Table 30: User-agents Legend

Code	User-agent
(A)	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.116 Safari/537.36 Edge/15.15063
(B)	Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko
(C)	Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko
(D)	Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3163.100 Safari/537.36
(E)	Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0)
(F)	Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/53.0.2785.116 Safari/537.36
(G)	Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/53.0.2785.116 Safari/537.36
(H)	Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:49.0) Gecko/20100101 Firefox/49.0
(I)	Mozilla/5.0 (iPhone; CPU iPhone OS 11_3 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.0 Mobile/15E148 Safari/604.1
(J)	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3071.115 Safari/537.36
(K)	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/67.0.3396.87 Safari/537.36 OPR/54.0.2952.64
(L)	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/68.0.3440.106 Safari/537.36
(M)	Mozilla/5.0 (iPhone; CPU iPhone OS 11_4_1 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.0 Mobile/15E148
(N)	Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/68.0.3440.106 Safari/537.36
(O)	Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:62.0) Gecko/20100101 Firefox/62.0
(P)	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/68.0.3440.106 Safari/537.36
(Q	Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/67.0.3396.87 Safari/537.36
(R)	Mozilla/5.0 (iPhone; CPU iPhone OS 11_4_1 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/11.0 Mobile/15E148 Safari/604.1
(S)	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.181 Safari/537.36
(T)	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/64.0.3282.140 Safari/537.36 Edge/17.17134
(U)	Mozilla/5.0 (iPhone; CPU iPhone OS 11_4_1 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Mobile/15G77
(V)	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/66.0.3334.0 Safari/537.36

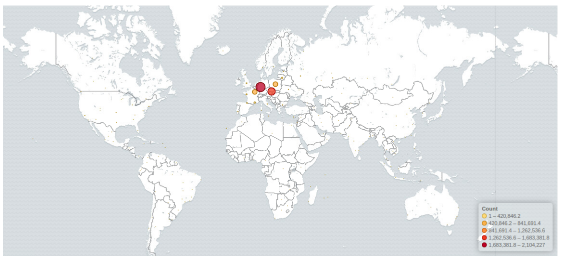
4.3.7 Diversity based on geolocation

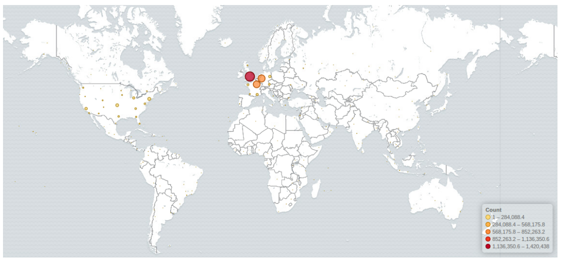
Figures 15 and 16 show the differences in geolocation we have found between D1 and D2. The same information is given for comparing D2 and D3, in Figures 17 and 18. The left-hand side of the figures show geolocation data relative to D1/D2, while the right-hand side of the figures show the geolocation data for D2/D3.

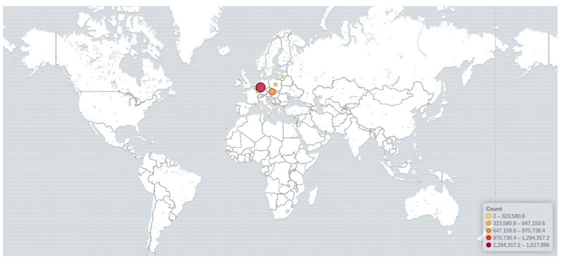
Comparing D1 and D2, for Arcane-only true positives and false positives, there seems to be no real difference between the two datasets. However, there are some striking differences between the datasets when it comes to CommTool-only positives. In particular, the most important difference concerns CommTool-only true positives: CommTool identifies bot traffic from a much wider geographical area compared with Arcane (which mainly reported alerts from Europe-based IPs, where the domain being monitored is located in). For D1, the true positives were mostly limited to European IPs, while in D2 they are found worldwide. We also see slight differences when it comes to CommTool-only false positives, with D1 having a bigger percentage concentrated in Central America, while in D2, these are concentrated throughout Europe. However, in these cases, the numbers are too small to derive any reasonable conclusions.

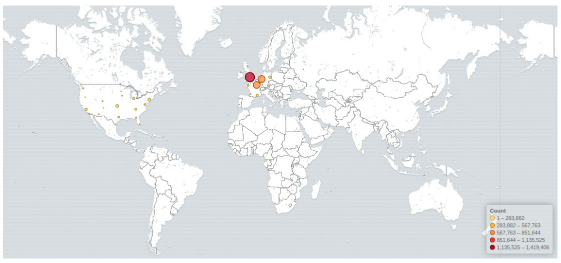
We speculate that the changes we observe are due to a change in botnet geographical dispersion between the two studies.

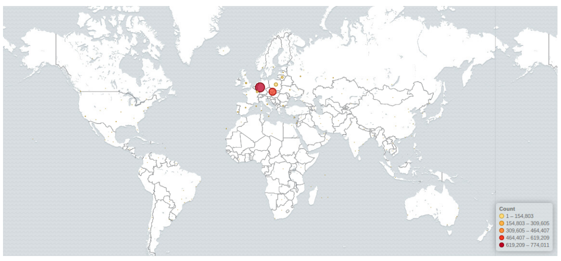
Comparing D2 and D3, which were collected from different domains, we see that, with the exception of CommTool-only true positives, the first domain appears to be targeted mainly by European addresses, while the second domain shows a much wider range of attacks, in terms of their geographical distribution. This makes sense because, as we noted before, the first dataset pertains to a European client while the second dataset pertains to a south Asian client.

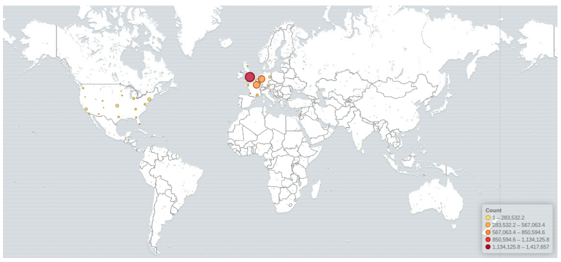

(a) All attacks (D1)

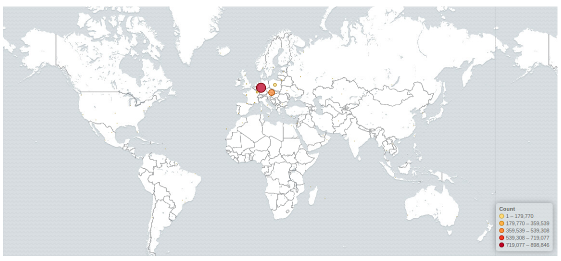

(b) All attacks (D2)

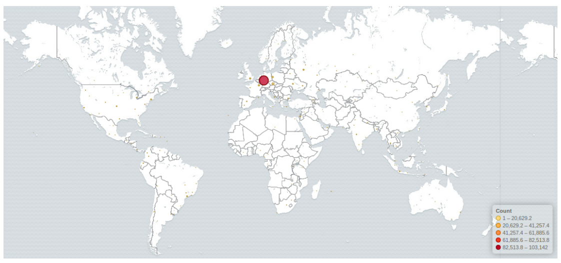

(c) Arcane true positives (D1)

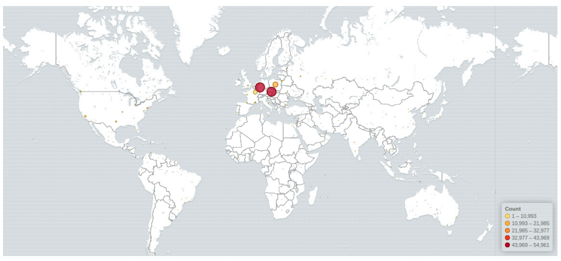

(d) Arcane true positives (D2)


(e) CommTool true positives (D1)


(f) CommTool true positives (D2)


(g) Arcane-only true positives (D1)


(h) Arcane-only true positives (D2)


(i) CommTool-only true positives (D1)

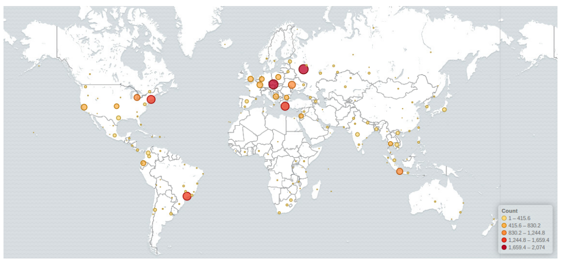
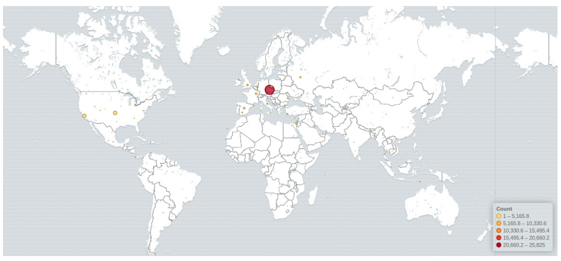
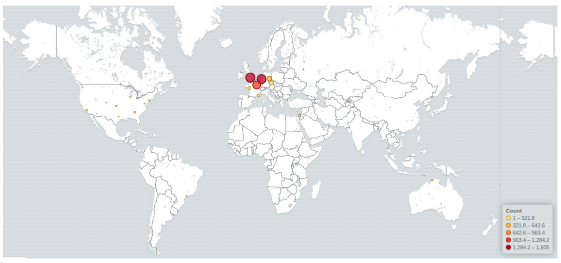

(j) CommTool-only true positives (D2)

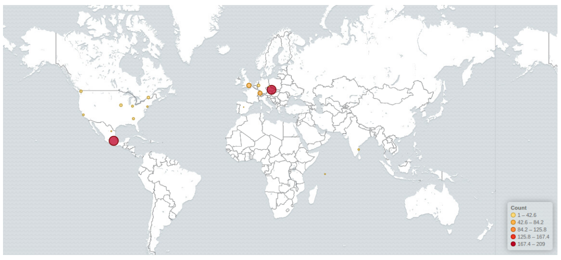
Figure 15: Geolocation of all attacks and true positives for D1 and D2



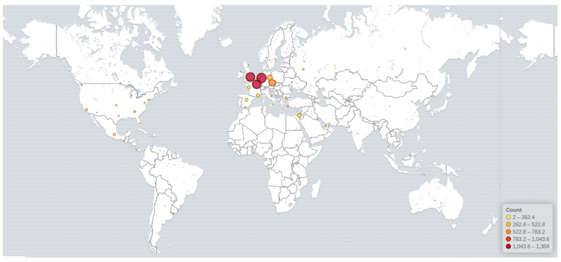
(a) Arcane false positives (D1)



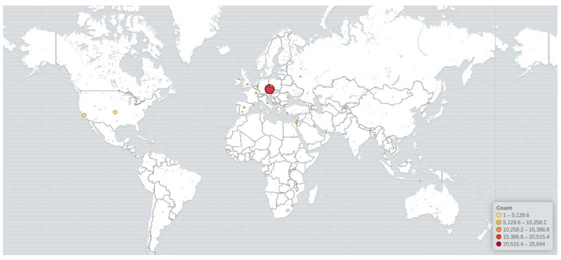
(b) Arcane false positives (D2)



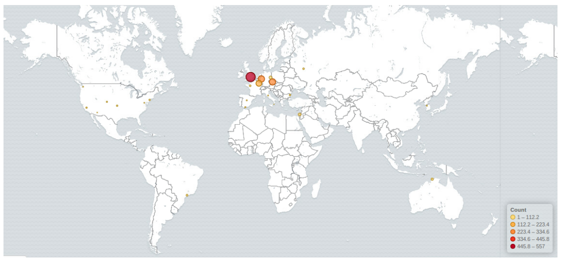
(c) CommTool false positives (D1)



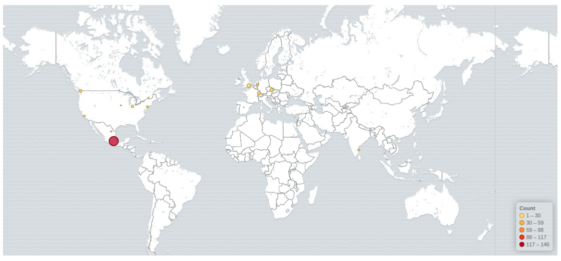
(d) CommTool false positives (D2)



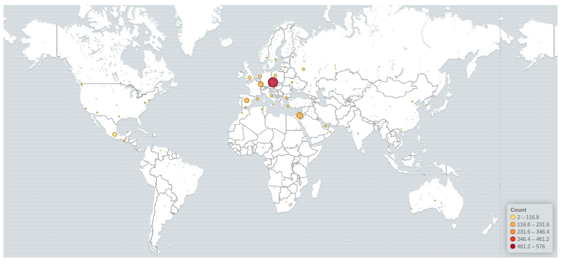
(e) Arcane-only false positives (D1)



(f) Arcane-only false positives (D2)

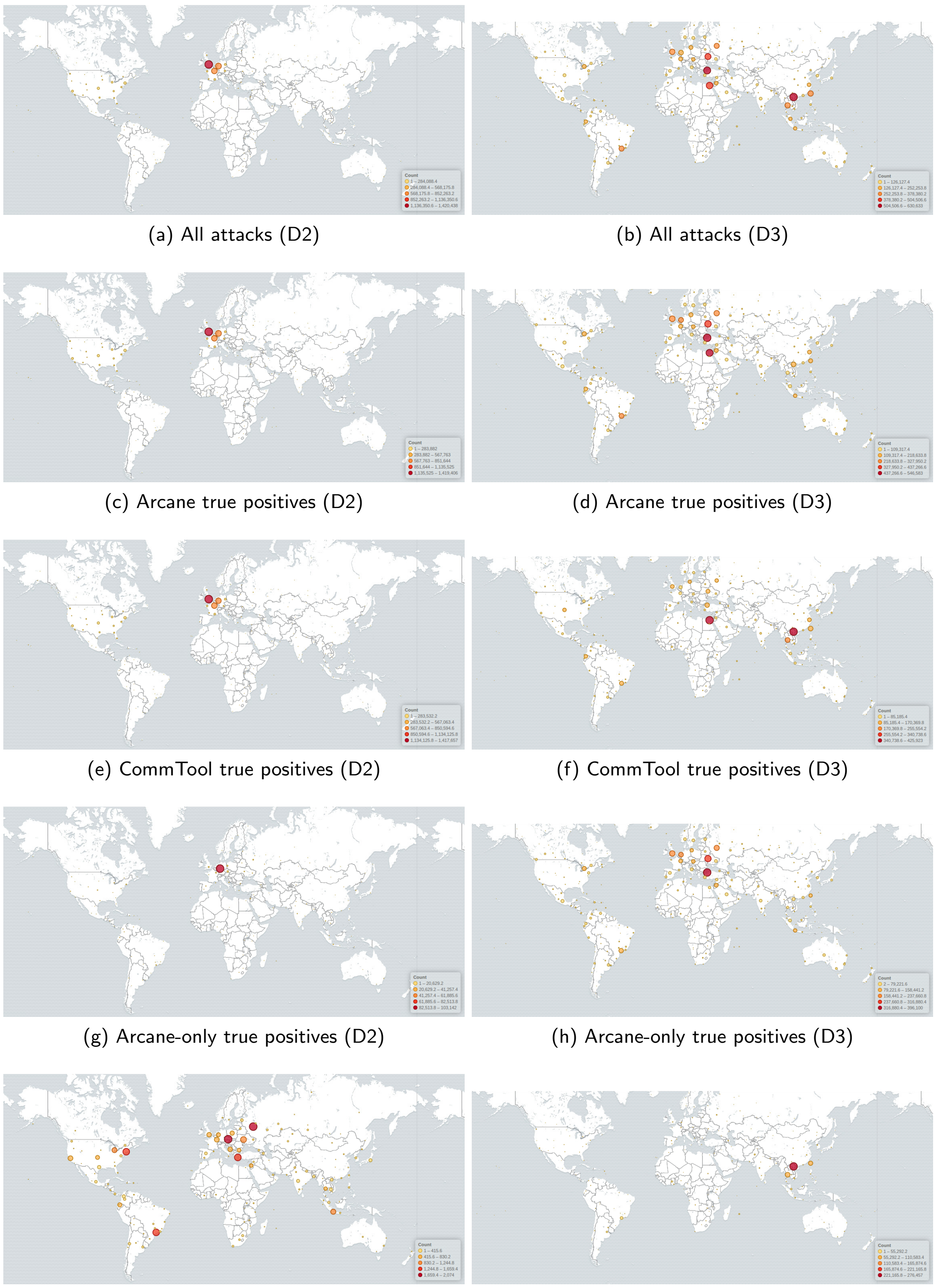


(g) CommTool-only false positives (D1)



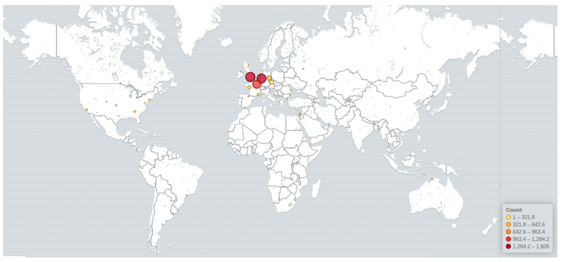
(h) CommTool-only false positives (D2)

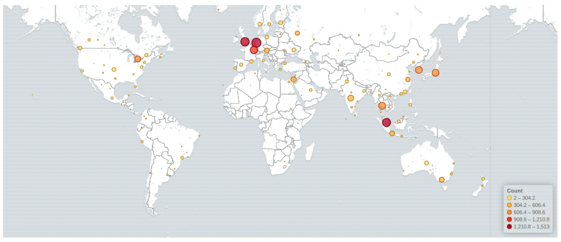
Figure 16: Geolocation of false positives for D1 and D2

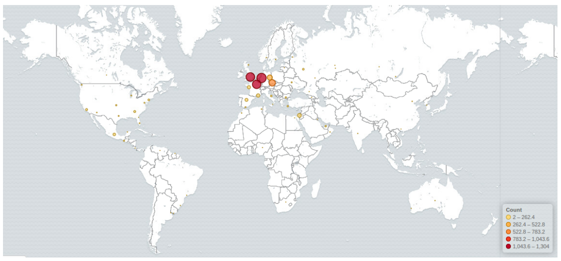


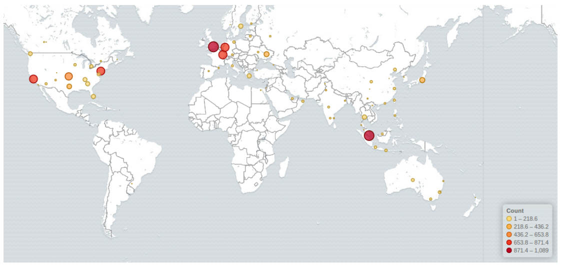
(a) All attacks (D2)
(b) All attacks (D3)
(c) Arcane true positives (D2)
(d) Arcane true positives (D3)
(e) CommTool true positives (D2)
(f) CommTool true positives (D3)
(g) Arcane-only true positives (D2)
(h) Arcane-only true positives (D3)
(i) CommTool-only true positives (D2)
(j) CommTool-only true positives (D3)

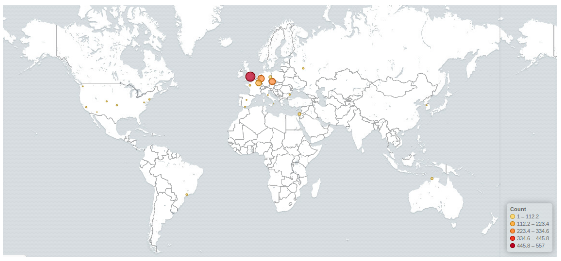
Figure 17: Geolocation of all attacks and true positives for D2 and D3

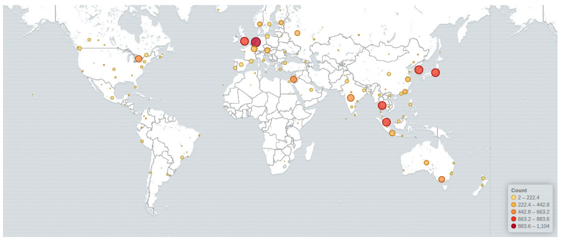

(a) Arcane false positives (D2)

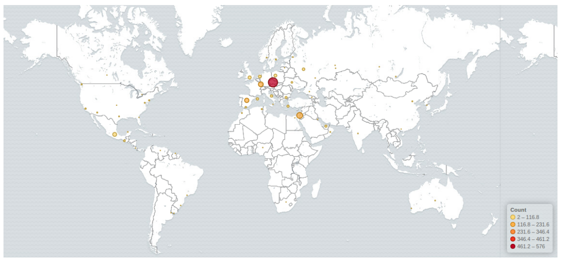

(b) Arcane false positives (D3)


(c) CommTool false positives (D2)


(d) CommTool false positives (D3)


(e) Arcane-only false positives (D2)


(f) Arcane-only false positives (D3)


(g) CommTool-only false positives (D2)

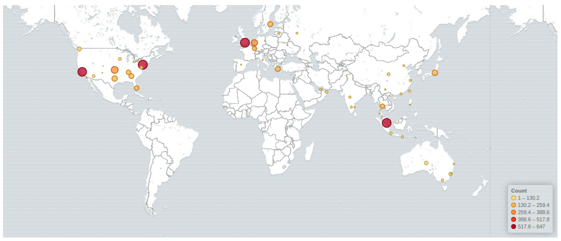

(h) CommTool-only false positives (D3)

Figure 18: Geolocation of false positives for D2 and D3

4.4 Conclusions

In our analysis, we've made a point of comparing D1 to D2 as two datasets gathered from the same domain but where the tools evolved over time, and comparing D2 to D3 as two datasets gathered from different domains but during the same period of time. Comparing D1 to D2, we show clear signs of diversity benefits for most setups, as evidenced by the values of sensitivity and specificity for 1oo2 and 2oo2 adjudication systems. Similar benefits in sensitivity and specificity are also seen in D3, however to a lesser degree.

There are two major factors for why we see differing alerting patterns between Arcane and CommTool. Firstly, the mode of operation between the two tools is substantially different from one another. CommTool works primarily on the client-side, and most of its detection capabilities are due to JavaScript tests run on the client's device. This translates to a generally faster decision as to whether a request is malicious or not but prevents CommTool from correlating between different connections. Arcane is exclusively server-side, and thus is capable of looking and correlating between multiple different client requests. This means that Arcane can more easily detect evidence of botnets. It also means that Arcane outputs more accurate alerts for longer connections. Secondly, the use of different known violator databases leads both tools to detect different connections. This, however, is not inherent to their operation modes.

In terms of the evolution of tools, we concluded that in D1 the tools complemented each other in their detection capability very well, though which adjudication scheme to use would depend on the losses associated with the different types of failures (false positives and false negatives). Using the results of the D1, the weaknesses of each tool were highlighted, and both tools were reconfigured by the dataset provider to improve their detection capabilities. As a result, the detection rate of each tool in D2 improved significantly. Hence, in D2, there is less room for improvements from diversity as the individual tools are already very good. Nevertheless, we still observed diversity in the tools' behaviour even in D2.

Similar conclusions can be drawn between D2 and D3, in terms of the transferability of tools to different domains, however, the benefits of design diversity were much smaller in both of these compared to D1. A large portion of the differences between D2 and D3 related directly to the different domains being analysed, although the improvements of design diversity are consistent between the two domains.

The nature of the datasets analysed limits us from making more general comments on the usefulness of design diversity when it comes to the detection of malicious web scraping activity. Our observations are applicable to the tools in our datasets, but it is hard to relate the benefits we saw in our analysis to how other web scraping detectors would operate.

5 Diversity with RNN malware detection

In this chapter, we present our analysis of the potential diversity benefits of using multiple recurrent neural network (RNNs) models for the classification of malware and benignware samples. We used a dataset from a previously published research [12], which contains predictions from a set of 37 RNN models, trained and tested on a dataset of over 4000 software samples. We have analysed the potential diversity benefits in combinations of up to 10 models, in 1-out-of-N, N-out-of-N and simple majority adjudication schemes.

This chapter is outlined as follows: in section 5.1 we go into more detail on the work that this analysis is based upon, including the building, training and testing of the various machine learning models analysed; section 5.2 describes the methodology we have followed to carry out our research; in section 5.3 we give an initial overview of the diversity present between the various models, and follow this in section 5.5 where we attempt to explain where this diversity stems from; finally, in section 5.6 we conclude our analysis. The application of optimal adjudication analysis on this dataset is presented separately, in Section 6.

5.1 Background work

The analysis presented in this chapter follows on the work done by Rhode et. al. [12]. For that work, the authors built and trained a set of 37 recurrent neural network (RNN) models for use in the classification of software samples as either malicious or benign. This work differs from others in the same context by utilising dynamic data analysis. Traditionally, when classifying software samples, static code analysis techniques are often the go-to approach when building reliable solutions. This is the approach that antiVirus products normally take, where code is extracted from the sample being scrutinised, and these samples are then compared to previously known malicious signatures. However, such approaches can fail to detect malware samples in one of two major ways: i) when encountering zero-day vulnerabilities, meaning that their signatures have not been indexed yet [19]; ii) when dealing with code obfuscating techniques that mask known malware signatures [110].

Dynamic data analysis techniques are an attempt to solve both of these issues, and are based on the idea that malware cannot avoid leaving traces of its malicious activity. The analysis of the activity generated by a sample run on a system (typically in a virtualised environment) can offer insight into the ultimate alignment of the code and thus allows for the detection of malware. These dynamic approaches often yield more reliable results than those found with static code analysis ([111–113]), however, they suffer from having to wait for potential malware to run, imposing a time penalty on the end-user or allowing a malware to inflict damage upon a system before it is detected and stopped.

The work by Rhode et. al. that we are expanding upon proposes a dynamic data analysis approach using recurrent neural network models (RNN), and attempts to generate predictions within the first few seconds of a sample being executed, thus lowering the typical time penalty of related work to a tolerable amount. The intuition for this is based on the idea that malware samples will typically attempt to perform their malicious actions as soon as possible, in order to shorten the time window for their attack and thus potentially avoid detection. For this, the ability for models to process time-series data (the data generated by a sample during the time it is ran) is necessary, and two major machine learning models are useful for this - RNN and Hidden Markov Models (HMM). The justification behind the use of RNN rather than HMM models is that the former are capable of processing *continuous* time series data, unlike HMMs.

A total of 37 RNN models were created. Each model works by analysing the activity generated by a sample when run in a virtualised environment, using 10 different metrics as feature inputs: system CPU usage, user CPU usage, packets sent, packets received, bytes sent, bytes received, memory use, swap use, the total number of processes currently running, and the maximum process ID assigned. A snapshot of the metrics is taken every second for 20 seconds whilst the sample executes, starting at 0 seconds, such that at 1 second, there will be two feature sets or a sequence length of 2.

Whilst API calls to the operating system are the most popular behavioural feature used in dynamic malware detection, there are some major reasons they were not used in this study as recent work has shown that they achieve comparable performance to the machine metrics used in this paper but are a less robust data source when tested on data from a different underlying distribution [114]. From a practical standpoint, there is also the issue that incorporating categorical features (which API calls are) requires an input vector with a placeholder for each category to record whether that API call was present or not. Due to the large amount of API that can be collected, a very large input vector would in turn be necessary, leading to a slower training process for the models.

In order to create RNN models that are diverse from one another, each model was built with a different set of hyperparameters. Table 31 describes these hyperparameters in more detail. When creating the models, a random search of the hyperparameter space was used as this allowed for easier parallelisation and implementation. Additionally, random space search has been found to be more efficient at finding good configurations when compared to grid search [115], which uses a discrete search space for hyperparameter values rather than a continuous one. This efficiency and automated nature are crucial, as Rhode hypothesises that the various classification models would need to be regularly re-trained as malware evolves and new samples are discovered.

To be able to generate predictions for the entirety of the sample dataset, while still making use of this same dataset for training purposes, a 10-fold cross validation technique was used. The sample dataset was split into 10 equally sized partitions. Models were trained with 9 of these and tested on the remaining partition, a process which was repeated 10 times, each time changing which partition was tested. The predictions generated for the 10 testing partitions were then combined, for ease of performance comparison between different models. It is crucial to note that when generating a prediction for a sample, a model was never trained using that same sample. The predictions generated by the RNN models are expressed in values between 0 and 1, where a prediction of 0.5 or higher is interpreted as a malware classification, with anything under this value being considered a benignware classification.

Table 31: Model hyperparameters

Attribute	Description
Depth	Number of hidden layers in GRU-RNN
Hidden neurons	Number of neurons (GRU-cells) in each hidden layer
Bidirectional	Time series processed forwards as well as backwards
Batch size	Number of training samples seen by the network between each weight update
Epochs	Number of times model is exposed to full dataset
Optimiser	Weight update rule
Learning rate	Multiplier on weight changes during training for SGD (0.001 learning rate used for Adam)
Dropout rate	Proportion of randomly zero-ed neurons during training to help with overfitting
L1 recurrent weight regulariser	L1 normalisation on weights
L2 recurrent weight regulariser	L2 normalisation on weights
R1 recurrent weight regulariser	R1 normalisation on weights
R2 recurrent weight regulariser	R2 normalisation on weights
Sequence length	Amount of data seen by each model for each behavioural trace

The sample dataset contained a total of 4066 software samples, of which 1925 were malicious and another 2141 were benign. The malware and benignware were collected from a variety of different online sources, including VirusTotal, Softonic, PortableApps, SourceForce[11] and Windows OS. Labelling of the software samples was done as a combination of manual labour and the use of the VirusTotal API. Additionally, VirusTotal results were used to classify malware samples based on the category of malware they resembled (counts of malware types are shown in Table 32). Each sample was run in a virtualised environment, using Cuckoo Sandbox, and machine activity metrics extracted using a custom auxiliary module reliant on the Python Psutil library. This is illustrated in Fig. 19.

The predictions generated by the models individually for each of the samples was published in [12]. In that publication the authors achieve a detection accuracy of 94% with just 5 seconds of dynamic data analysis, compared to a typical file execution time for dynamic analysis being around 5 minutes. In our research, we expanded this analysis by looking at the possible gains and drawbacks of using multiple RNN models in more conventional adjudication schemes, as we have done with our work on the web scraping detector analysis.

[11] www.virustotal.com; www.softonic.com; www.portableapps.com; www.sourceforge.net

Table 32: malware type counts

Type	Count
trojan	1056
virus	309
backdoor	114
adware	86
trojanransomware	72
bot	71
virusransomware	52
adwareransomware	50
application	36
worm	24
aptvirus	20
infostealertrojan	16
rootkit	11
aptbackdoor	7
unknown	1

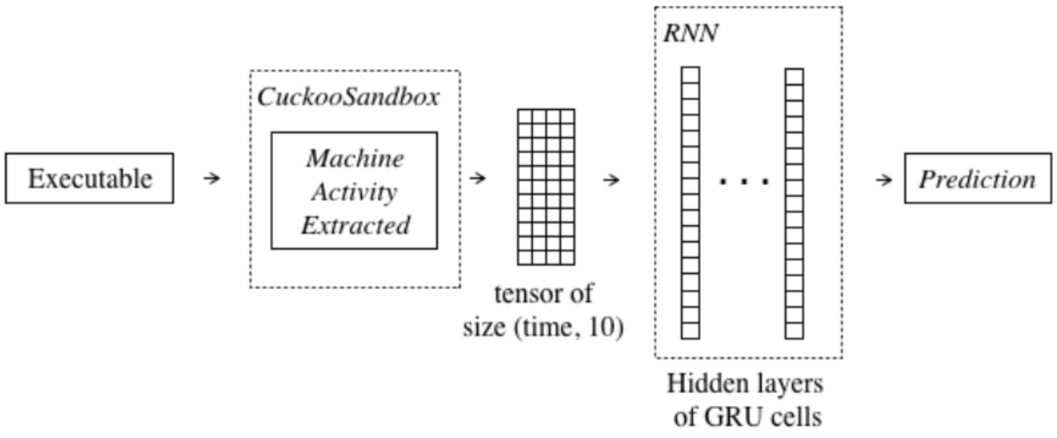

Executable
CuckooSandbox
Machine Activity Extracted
tensor of size (time, 10)
RNN
Hidden layers of GRU cells
Prediction

Figure 19: High-level model overview

5.2 Methodology

Our analysis is based on the predictions generated by the 37 RNN models created in the initial publication, for all samples in the original dataset (malicious and benign). These predictions were in the form of a numerical value between 0 and 1, where a prediction of 0.5 or above indicated that the model predicted that the sample was malicious. By looking at the outputs generated by a combination of models for all of the samples analysed, we are able to extrapolate what their combined output would be under different adjudication schemes, thus allowing us to calculate various performance metrics (such as accuracy, sensitivity and specificity).

We have done this for combinations of up to 10 models[12] in three different types of adjudication schemes: i) 1ooN schemes, which improve sensitivity; ii) NooN schemes, which improve specificity; and iii) simple majority schemes (i.e. 2oo3, 3oo5, 4oo7, 5oo9) which represent a middle ground between sensitivity and specificity.

Additionally, we have done this analysis based on the various hyperparameter values of each model, allowing us to better understand which parameters lead to an increased diversity between models, as well as based on malware type, making it possible to determine whether design diversity techniques have different effects for different types of malware.

5.3 Diversity results

We first look at how all of the models in our dataset performed on their own. In Fig. 20 we show the accuracy, sensitivity, and specificity that all 37 models achieved during their testing, ordered by their overall accuracy. Most of the models achieve close to 0.7 accuracy, with a few outliers on either side of the graph. However, the same cannot be said for sensitivity and specificity, as these vary significantly between the models, with some being better at detecting malware, while others better at not raising false alarms for benignware. The three worst models (models 29, 24 and 6) all vary significantly from the rest of the models, by having their sensitivity close to 0.2, while model 1 (being the best in overall accuracy) is the only model to achieve a higher value for sensitivity then specificity.

When looking at these graphs it is important to keep in mind the various parameters with which the models were built. The three worst models from the previous figure (models 29, 24 and 6) are the only models in our dataset that have a value of 20 for sequence length (the amount of time models look at a sample's activity before generating a prediction), which is also the highest value of sequence length in the dataset. On the other hand, the best performing model (model 1) is the only model in the dataset with a sequence length of 2, which also happens to be the lowest value of sequence length in the dataset. If we order the information we have by the value of each model's sequence length, as we have done in Fig. 21, we can see that as the value of sequence length increases, there is a consistent decrease in that model's sensitivity, with an opposite but subtle increase in specificity. It's important to note that if a malware sample did not run for as long as the sequence

[12]We ran the analysis in a distributed computational environment at City, University of London, which utilised three VMs, each with 20 CPU cores, and with 64GB RAM in each VM. It took approximately 10 days to run the 1oo10 experiment, and, due to the combinatorial explosion, we estimated it would take more than a month to do 1oo11, and even longer for higher combination sizes. For this reason, we did not continue with higher combinations.

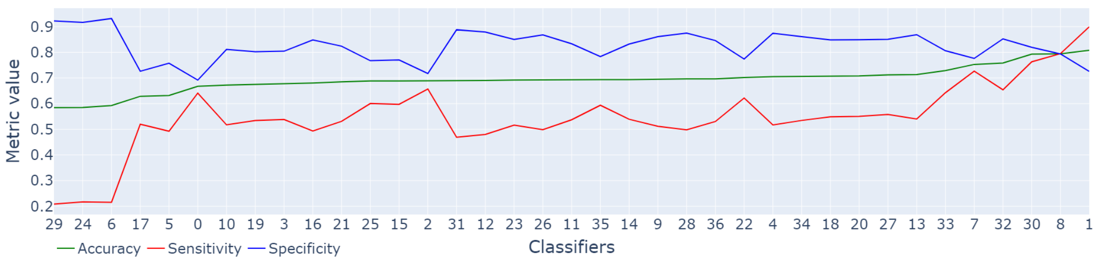


Figure 20: Single model metrics, ordered by accuracy

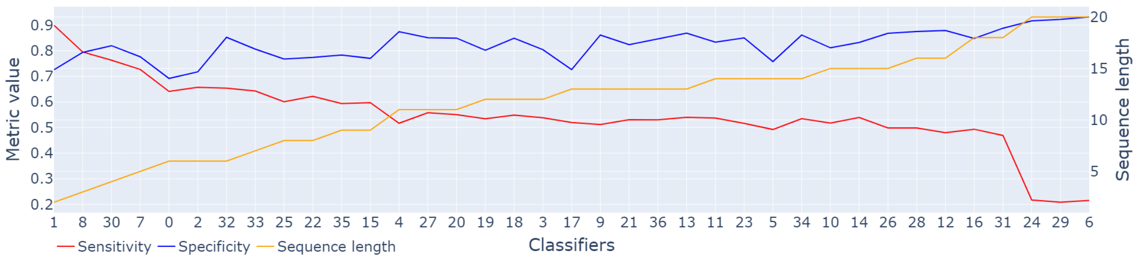


Figure 21: Single model metrics, ordered by sequence length

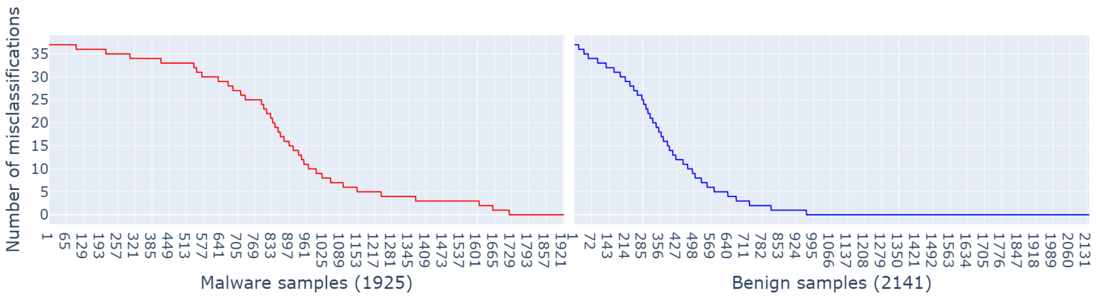


Figure 22: Sample difficulty for malware and benign samples

Table 33: Accuracy, sensitivity and specificity values for single models

Model	**Accuracy**	**Sensitivity**	**Specificity**
1	0.80792	0.89922	0.72583
8	0.79415	0.79481	0.79355
30	0.79292	0.76312	0.81971
32	0.75848	0.65403	0.85241
7	0.75283	0.72675	0.77627
33	0.72873	0.6426	0.80617
13	0.71323	0.54026	0.86875
27	0.712	0.55792	0.85054
20	0.70782	0.55065	0.84914
18	0.70659	0.54857	0.84867
34	0.70659	0.53455	0.86128
4	0.70512	0.51688	0.87436
22	0.70192	0.62182	0.77394
36	0.69651	0.53039	0.84587
28	0.69651	0.49818	0.87482
9	0.69577	0.51169	0.86128
35	0.69356	0.59377	0.78328
14	0.69356	0.53922	0.83232
11	0.69306	0.53714	0.83326
26	0.69306	0.4987	0.86782
23	0.69208	0.51636	0.85007
12	0.69011	0.48	0.87903
31	0.68962	0.46909	0.8879
2	0.68888	0.65714	0.71742
15	0.68839	0.5974	0.7702
25	0.68839	0.60052	0.7674
21	0.68519	0.53091	0.82391
16	0.68028	0.49351	0.8482
3	0.67831	0.53818	0.8043
19	0.67511	0.53403	0.80196
10	0.67216	0.5174	0.8113
0	0.66773	0.64104	0.69173
5	0.63207	0.49247	0.75759
17	0.62863	0.52	0.7263
6	0.59272	0.21558	0.93181
24	0.58534	0.21714	0.91639
29	0.58436	0.20831	0.92247

length value of a model, that model did not generate a prediction since it did not have the "necessary" amount of information needed to do so. In these cases, when calculating the performance of these specific models, those samples which ran for less than the required sequence length were excluded. This means that the metrics for higher sequence length models are less accurate compared to lower sequence length models.

Of all the hyperparameters used when building the models, sequence length is the only one that seems to have a demonstrable impact on the various metrics achieved by the individual models. For the reader's insight, we show in Table 33 all of the models in the dataset alongside their achieved accuracy, sensitivity and specificity.

It is also important to understand the difficulties that each sample in our dataset poses to the models. Fig. 22 represents the "difficulty" of each sample, by identifying the total number of models that failed to correctly identify the various malware and benignware samples (i.e., they labelled malware as benign and benign as malware). An important aspect to highlight in each of the lines is that, at times, they reach a total of 37 models, meaning that there are both malware and benignware samples that are always incorrectly classified by all the models in our dataset. This effectively imposes certain upper bounds on the maximum amount of sensitivity and specificity we can ever achieve with just combinations of these same models, something we will touch on in the next section[13]. For malware, there are a total of 101 samples that are never correctly identified as malware, which means that the upper bounds for sensitivity with conventional adjudication schemes is (1925 - 101) / 1925 = 0.94753. For benign samples, there are a total of 18 samples that are never correctly identified as benign, which means that the upper bound for specificity with conventional adjudication schemes is (2141 - 18) / 2141 = 0.99159.

Of the 101 malware samples that were always misclassified, there doesn't appear to be a clear reason, other than their intrinsic difficulty. In Table 34 we break down these 101 samples based on their malware type, in which trojan and virus are the most numerous. However, this is likely due to the distribution of overall malware types in the sample dataset, as this same order is found when looking at all of the malware samples in the dataset (as we previously demonstrated in Table 32).

5.4 Overview of diversity

Now that we have analysed the individual models, we can start to combine them in different adjudication schemes to try to understand what sort of benefits and drawbacks we can get, in terms of overall accuracy, as well as sensitivity and specificity. We have looked at 1-out-of-N schemes which excel at improving the sensitivity of the system (correctly identifying malware), N-out-of-N schemes which excel at improving the specificity of the system (correctly identifying benignware) and simple majority schemes, which are often a mix of the best of both worlds.

In Figs. 23 and 24 we show the sensitivity and specificity improvements with 1ooN and NooN adjudication schemes, respectively, up to and including combinations of 10 models. As expected, each adjudication strategy improves the respective metric of the

[13]Higher values of sensitivity and specificity may be achieved using optimal adjudication schemes. We will discuss this in chapter 6

Table 34: Description of malware never identified

Type	Count
trojan	56
virus	19
backdoor	9
adwareransomware	4
bot	3
trojanransomware	3
adware	2
infostealertrojan	2
aptvirus	1
virusransomware	1
worm	1

overall system, with 1ooN improving sensitivity and NooN improving specificity. For 1ooN (Fig. 23), there is a gradual increase in sensitivity up until 1oo4 schemes, with two more noticeable jumps after this point, going from 1oo5 to 1oo6, and from 1oo7 to 1oo8, as noted by the jump in the median.

For NooN (Fig. 24) we see a similar picture for specificity, with more significant improvements to the metric going up to 4oo4 schemes, and subtly continuing thereafter.

In Tables 35 and 36 we show the minimum, mean and maximum values for sensitivity and specificity for 1ooN and NooN adjudication schemes, respectively, along with the upper bounds for these metrics, as we calculated previously when looking at the sample difficulty. In accordance with the data in the previous boxplots, these metrics have a consistent growth as the number of models in each combination increases. For sensitivity, the maximum theoretical value is achieved with a combination of 1oo8. The maximum sensitivity is thus not increased for 1oo9 and 1oo10 combinations, but the minimum and mean sensitivity values continue to grow.

For specificity however, it's theoretical maximum value is not observed in the current analysis we have performed, with 10oo10 schemes only reaching a maximum of 0.98926 (differing from the upper bound by only 0.00233). The upper bound of specificity concerning this dataset would only be achieved with a minimum adjudication of 15oo15[14].

We have also looked at majority voting schemes. As a mix between 1ooN and NooN schemes, majority voting tends to strike a balance between sensitivity and specificity, leading to an increase in accuracy. In Figs. 25 and 26, we show the improvements for all three of these metrics when looking at majority voting schemes. For all three metrics, the effects of majority voting seem to be restricted to narrowing the range of possible values to the original median found in 1oo1. This significantly reduces the lowest minimums found in some combinations, but at the same time it also limits the outliers that overachieve.

[14]We were able to verify this using a separate set of scripts that looked only at combinations for which all the benign samples were correctly identified as benign, rather than calculating all possible combinations.

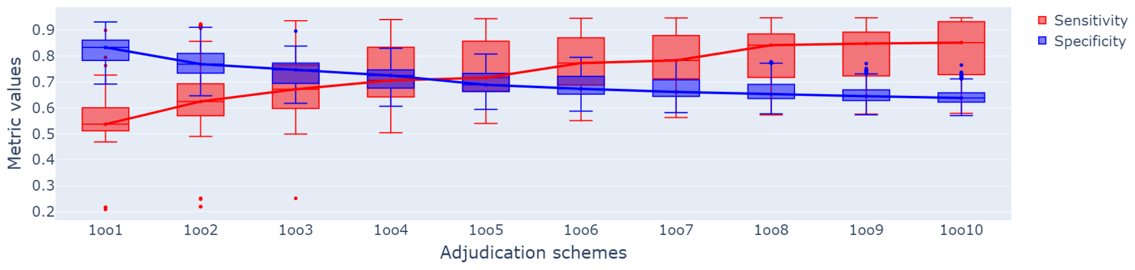


Figure 23: Sensitivity and Specificity for 1ooN adjudication schemes

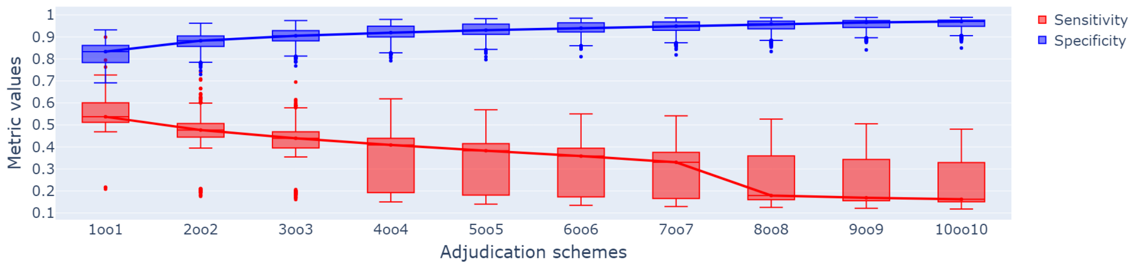


Figure 24: Sensitivity and Specificity for NooN adjudication schemes

Table 35: Min, Mean and Max sensitivity for 1ooN adjudication schemes

Adjudication	**Combinations**	**Min**	**Mean**	**Max**	**Upper bound**
1oo1	37	0.20831	0.54836	0.89922	0.94753
1oo2	666	0.21922	0.64942	0.92364	0.94753
1oo3	7,770	0.25143	0.69698	0.9361	0.94753
1oo4	66,045	0.50442	0.72969	0.94078	0.94753
1oo5	435,897	0.54026	0.75519	0.9439	0.94753
1oo6	2,324,784	0.55117	0.7761	0.94597	0.94753
1oo7	10,295,472	0.56312	0.79373	0.94701	0.94753
1oo8	38,608,020	0.57299	0.8089	0.94753	0.94753
1oo9	124,403,620	0.5761	0.82213	0.94753	0.94753
1oo10	348,330,136	0.57922	0.83381	0.94753	0.94753

Table 36: Min, Mean and Max specificity for NooN adjudication schemes

Adjudication	**Combinations**	**Min**	**Mean**	**Max**	**Upper bound**
1oo1	37	0.69173	0.82452	0.93181	0.99159
2oo2	666	0.72957	0.88045	0.96217	0.99159
3oo3	7,770	0.76833	0.90511	0.97431	0.99159
4oo4	66,045	0.79169	0.92058	0.97992	0.99159
5oo5	435,897	0.79682	0.93176	0.98319	0.99159
6oo6	2,324,784	0.81037	0.94041	0.98505	0.99159
7oo7	10,295,472	0.81831	0.94737	0.98645	0.99159
8oo8	38,608,020	0.83372	0.95312	0.98739	0.99159
9oo9	124,403,620	0.8412	0.95794	0.98832	0.99159
10oo10	348,330,136	0.8496	0.96203	0.98926	0.99159

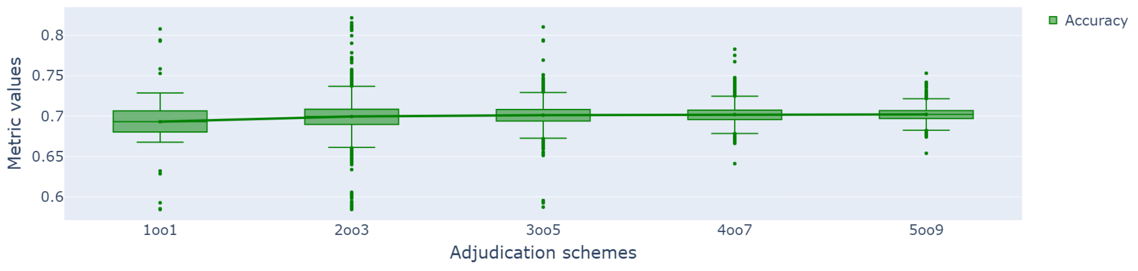


Figure 25: Accuracy for majority voting adjudication schemes

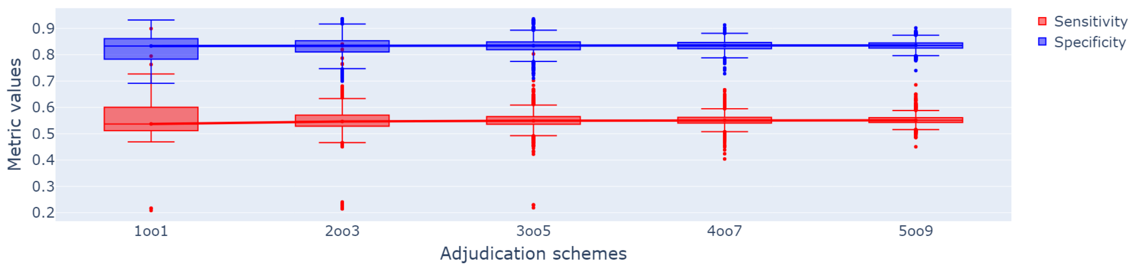


Figure 26: Sensitivity and Specificity for majority voting adjudication schemes

5.5 Sources of diversity

In the previous section we showed the analysis of the observed diversity when building combinations of N models. The question then arises as to what the source of the diversity between the various models is. We analysed all of the hyperparameters (cf. Table 31) to check for the effects they have on the observed diversity. Two of these hyperparameters appear to have the largest effect on the diversity between the models, namely the model's sequence length parameter, and the model's optimiser, both of which we will present in this section. For all other hyperparameters the differences they produce in the performance of combinations of different sizes are negligible or at best, random.

We start by looking at the effects of sequence length, which is the amount of time a model looks at a sample's activity before it generates a prediction. As we previously pointed out, when looking at individual models, an increase in the model's sequence length clearly accompanied a decrease in sensitivity for that model. In Figs. 27 and 28, we show the mean sensitivity and specificity for combinations in 1ooN and NooN schemes, respectively, based on the mean distance in sequence length of all the constituent models. The various model combinations are grouped together by looking at the individual sequence length value of the individual models in that combination, and calculating the mean difference between all of them. In essence, this means that combinations with a higher mean distance in sequence length have their constituent models more evenly spaced out in the range of all possible values of sequence length.

One thing that is clear from looking at Figs. 27 and 28 is that both sensitivity for 1ooN schemes, and specificity for NooN schemes increases as the mean distance of sequence length increases as well. This indicates that combinations with a wider range of sequence lengths are more "diverse". For 1ooN schemes, a more diverse combination leads to a higher number of positive (malware) samples being correctly identified, due to individual

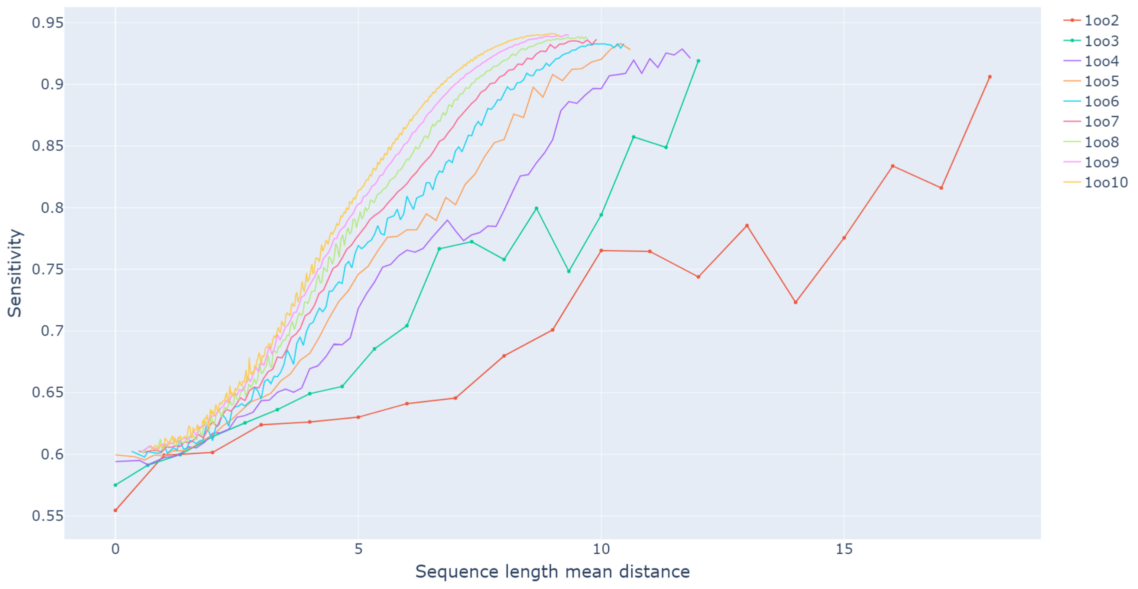


Figure 27: Mean sensitivity based on the mean distance between sequence length values of models in a particular combination

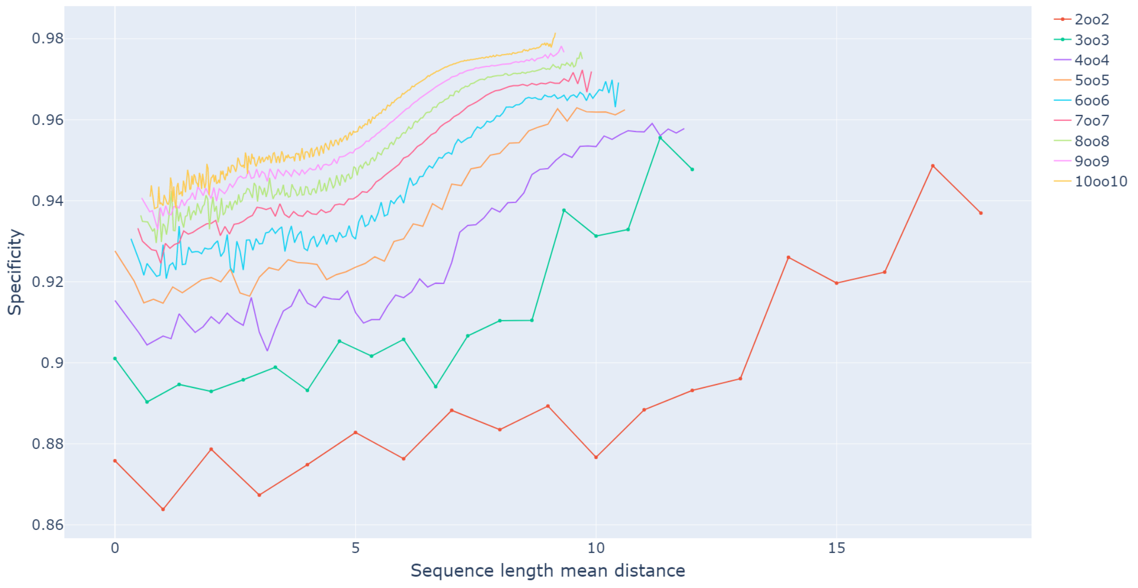
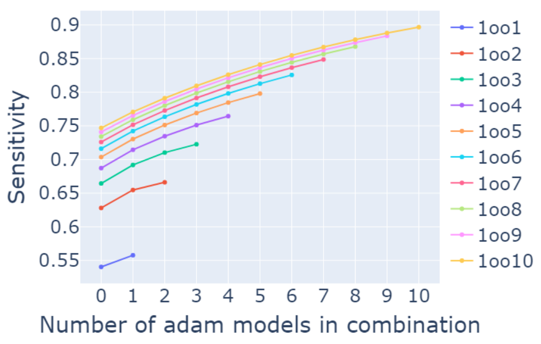


Figure 28: Mean specificity based on the mean distance between sequence length values of models in a particular combination

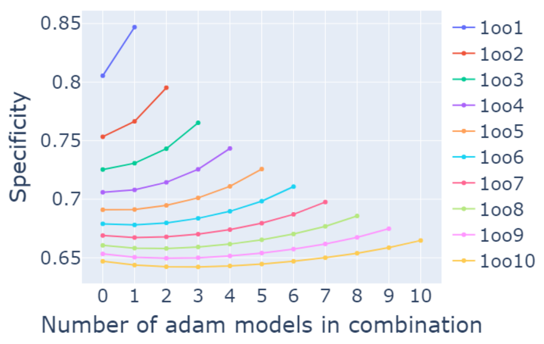
models alerting distinctly on different samples. For NooN schemes this same logic applies, but because all the models need to generate an alert for one to be raised overall, this ”diversity” between the individual models means that fewer false positives are generated.

One of the reasons that could explain these diversity gains coming from different sequence length values is the methods of attack of different malware samples. The patterns of attack of each sample will have a great effect on the activity observed by each model. For example, a malware might start generating suspicious activity as soon as it is run, such as creating a backdoor, but fall-off in activity afterwards, performing actions that may look more normal. In these cases, a model with a lower sequence length would have an advantage in its prediction, as it may only have seen the abnormal activity generated by the malware. The same logic could apply for slower acting malware, which could start off benign in order to mask their activity, and only generate more abnormal behaviour later on, meaning that a model with a higher sequence length would generally perform better. In situations where both types of malwares are to be expected, the use of different types of machine learning models would thus be a worthwhile advantage.

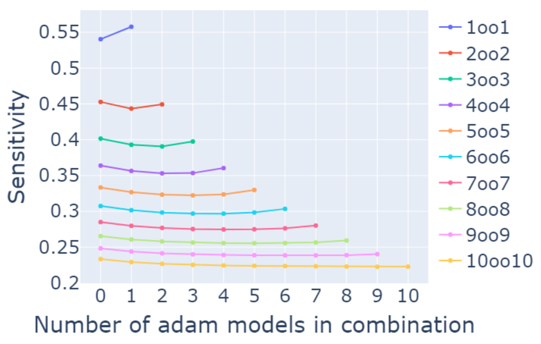
We now move on to the effects of the optimiser used for each model. Each model operates using one of two optimisers, either Adaptive Moment Estimation (Adam) or Stochastic Gradient Descent (SGD). SGD uses gradient descent to minimise error in machine learning, computing using just a subset of the data, but requires the model builder to choose a learning rate at which model parameters are updated. Adam has risen to popularity as it adapts the learning rate and typically performs well using an initial default learning rate. However, there is debate as to which yields a more accurate model [116]. In Fig. 29 we compare the mean sensitivity and specificity values for 1ooN and NooN schemes based on the various possible configurations of Adam and SGD models. We display this by plotting on the X-axis the number of Adam models present in the combinations, such that on the left most point of the graphs, where X = 0, there are only SGD models in the



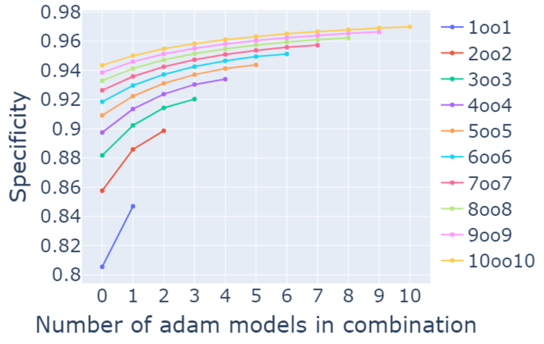
(a) Mean sensitivity based on optimiser combinations for 1ooN schemes



(b) Mean specificity based on optimiser combinations for 1ooN schemes



(c) Mean sensitivity based on optimiser combinations for NooN schemes



(d) Mean specificity based on optimiser combinations for NooN schemes

Figure 29: Mean sensitivity and specificity for 1ooN and NooN schemes based on optimiser combination (x-axis indicates the number of models in that combination using an Adam optimiser)

combination, and as we move right, we replace one of the SGD models with an Adam one.

The analysis shows that, for this dataset, the higher the number of Adam models in a combination, the better the performance of that combination. This is true for both sensitivity and specificity, in both 1ooN and NooN schemes, with the only exception being sensitivity in NooN schemes, where this only is true towards higher proportion of Adam vs. SGD model combinations. This aspect mostly highlights how Adam seems to be a better option when compared to SGD when it comes to applications of malware detection. This may be explained by the learning rates used by these algorithms which determines the amount by which to update the parameters of the neural network and may benefit from being large at the start of training, to allow big changes to the model parameters, and smaller later on, to make smaller adjustments once the model is performing reasonably well. Adam adapts the learning rate during training whilst SGD does not.

An interesting aspect to look at is the impacts of diversity on different types of malware samples. In Fig. 30, we show the improvements to sensitivity when going from 1oo1 to 1oo2 and 1oo3 schemes, broken down by malware sample types. Note that the data is ordered on the graph based on the improvement achieved when going from 1oo1 to 1oo3 schemes. For most malware types, going from 1oo1 to 1oo3 leads to close to 0.15 improvement in sensitivity. However, for some, like "application", this improvement can go as high as 0.22.

Additionally, we can further break down the mean sensitivity by the distance in sequence length properties of models, like we did previously in Fig. 27. In Fig. 31 we show the mean sensitivity achieved by combinations of models in 1oo3 schemes, based on the mean distance in sequence length for each of the malware types present in our dataset. As was the case when looking at the mean sensitivity of all possible combinations, a higher distance between the sequence length property of models in the same combination leads to an increase in sensitivity. Note that, for two types of malware, namely "aptbackdoor" and "rootkit", higher distances of sequence length actually achieve a sensitivity of 1, meaning that all samples of these types are correctly detected. Keep in mind however, that this might just be due to the small number of samples of these two types (11 and 7 samples respectively).

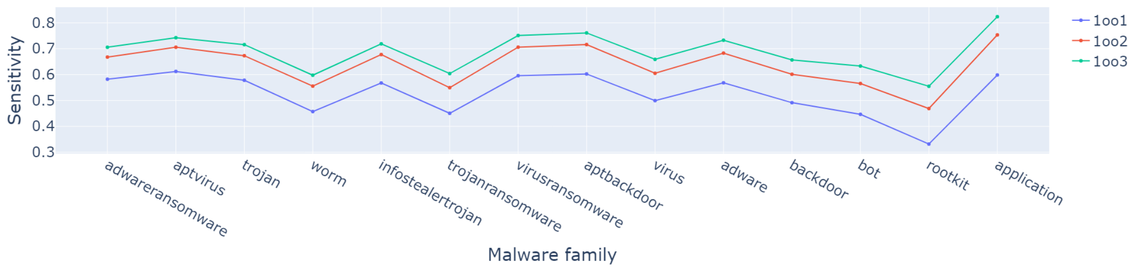


Figure 30: Mean sensitivity by malware type (ordered by largest improvement from 1oo1 to 1oo3)

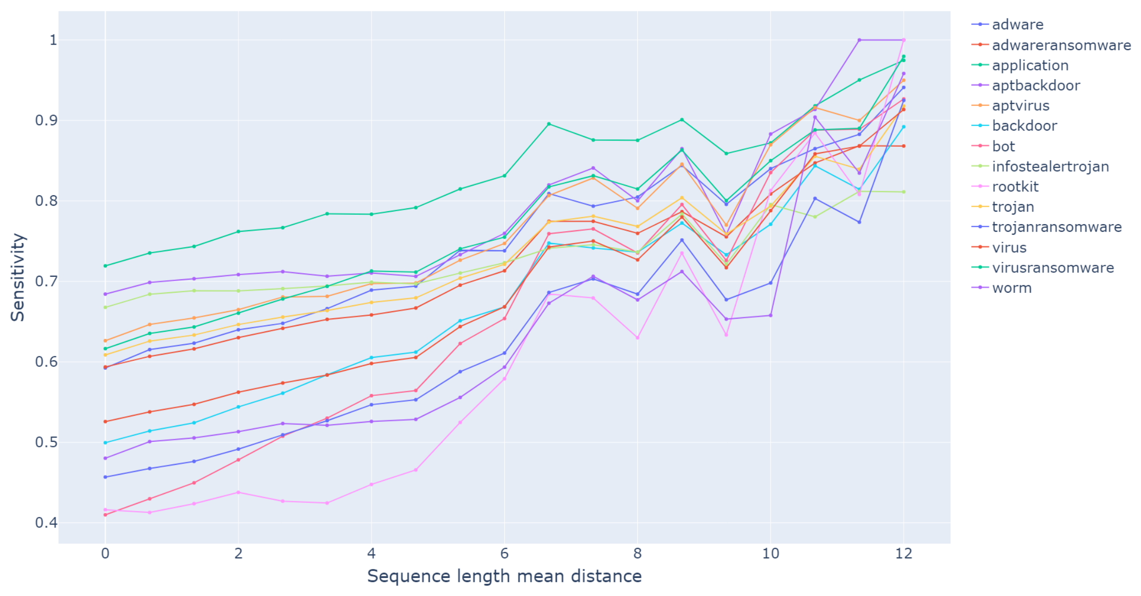


Figure 31: Mean sensitivity by malware type based on sequence length mean distance (1oo3)

5.6 Conclusions

In this chapter we have detailed the results of our research into the diversity benefits of combining multiple recurrent neural network models for the classification of malware samples. We describe the individual performance of the various models used in our analysis, and proceed to examine the benefits gained in terms of sensitivity, specificity and accuracy when we combine these models into 1-out-of-N, N-out-of-N and simple majority adjudication schemes.

The use of design diversity with this dataset yielded very positive results in regard to the models' major performance metrics. In terms of sensitivity, 1-out-of-10 schemes achieved a mean sensitivity of 0.83381, compared to the mean sensitivity achieved by single models of 0.54836 (an increase of 0.28545). The theoretical maximum value of 0.94753 was achieved in combinations as early as 1-out-of-8 (the theoretical maximum value is not 1 for our dataset because there are a number of malware samples that are never correctly classified as such). For sensitivity, 10-out-of-10 schemes also improved the mean value, going from 0.82452 in individual models to 0.96203 (an increase of 0.13751).

Our research highlights two major sources of diversity between the individual models, which are responsible for a great deal of benefits achieved from combining multiple models. The first of these is the amount of time a model has to analyse a sample before it should generate a prediction. A good mix of this ”sequence time” leads to a greater diversity in the alerts generated by the models. Secondly, a model's choice of optimiser seems to have a greater effect on the performance of models than any other hyperparameter being used. In our dataset, only Adam and SGD optimiser were used, and of these, combinations with more Adam models performed significantly better overall.

While the dataset we have analysed is fairly balanced in terms of the types of samples it contains, it's far from comprehensive. This limits the amount of confidence we can have on the conclusions we can make, especially when it comes to understanding the benefits of design diversity when tackling different types of malware samples. Additional work should be done with larger amounts of data, as well as different algorithms for machine learning models, as these can potentially present large diversity gains.

In the next chapter we will expand our research further, with the application of optimal adjudication on this study dataset, and its benefits when compared to more conventional adjudication schemes, as are the ones we have detailed in this chapter.

6 Optimal adjudication

In our previous two chapters we have looked at the benefits and drawbacks with the use of design diversity for the detection of malicious web scraping activity, and the use of machine learning for the classification of malware samples through the use of recurrent neural network (RNN) models. We did this solely with conventional adjudication schemes - i.e. 1ooN and NooN schemes. In this chapter we expand on both of those studies with the application of optimal adjudication techniques, an extension to the adjudication schemes we looked at previously.

This chapter is outlined as follows: in section 6.1 we detail the methodology of building an optimal adjudication; we then split our analysis into sections 6.2 and 6.3, where we detail the application of optimal adjudication techniques to both studies we have discussed in the previous chapters, respectively. In each of these sections we detail how we went about building the optimal adjudicators for each context and detail the results we observed; Finally, in section 6.4 we discuss the usefulness of these results, as well as the advantages of using optimal adjudication versus conventional adjudication schemes.

6.1 Methodology

We first discussed the idea of an optimal adjudication scheme in section 2.2.3. As opposed to conventional adjudication schemes where a quorum (of any size) is the deciding factor for an overall generated prediction (e.g., 1-out-of-N or N-out-of-N), an optimal adjudication scheme takes into account individual classifications of each classifier in the combination, and the unique combination in which these are generated. Each of these unique combinations of predictions is known as a *syndrome*, and these syndromes partition the sample space of our system into disjoint subsets, such that each sample triggers one, and only one syndrome. By partitioning the sample space of a classification system through these syndromes, we can determine what the best output should be for each syndrome in order to minimise cost, by simply looking at the number of positive and negative samples that are observed historically for each of them, and choosing the appropriate action (i.e., choosing to alert if there are more positives than negatives, or not alert otherwise). To better illustrate this idea, we bring back a similar example to that which we first introduced during our literature review chapter, in Table 37.

The number of total possible syndromes is an important aspect, as the number of syndromes dictates the granularity of the overall system. We can imagine a trivially simple system, one which does not contain any classifiers, meaning that we can define a single syndrome - a syndrome which encompasses the entirety of the sample space. In this example we can then decide whether to always alert, or never alert based on whether our system observes more positive or negative samples respectively. This will necessarily lead to a large number of false positives or false negatives. Once we add a single classifier to our system, we can then define two syndromes - one where our classifier generated an alert, and another where it didn't. Once again, we can look at each syndrome and determine whether to alert or not alert based on the number of positive and negative samples that triggered that syndrome (i.e. made our classifier raise an alert or not). The more classifiers we add to our system, and the more possible syndromes there are, the

Table 37: Optimal adjudicator example

	Positives	**Negatives**	**Optimal adjudication**
A = 0 **B = 0** **C = 0**	139	1418	**No alert**
A = 1 **B = 0** **C = 0**	20	107	**No Alert**
A = 0 **B = 1** **C = 0**	420	28	**Alert**
A = 0 **B = 0** **C = 1**	20	18	**Alert**
A = 1 **B = 1** **C = 0**	87	272	**No alert**
A = 1 **B = 0** **C = 1**	15	11	**Alert**
A = 0 **B = 1** **C = 1**	112	17	**Alert**
A = 1 **B = 1** **C = 1**	1112	270	**Alert**

smaller the syndromes that partition our sample space become. If we assume that there are some inherent similarities between malicious samples and between benign samples, and that these are different from one another, then more granular syndromes would mean that syndromes become more type heavy, meaning that we would usually find syndromes with many more of one sample type than the other, rather than syndromes where we have a close to 50-50 split. In turn, these lopsided syndromes mean that an overall system is able to generate less false positives and false negatives.

When applying optimal adjudication techniques to both of our previous studies (malicious web scraping detection and machine learning for software sample classification) we run into two different aspects of optimal adjudication. When dealing with malicious web scraping detection, we are faced with a binary classification system, where either classifier outputs a 0 (benign) or a 1 (malicious) when asked to classify a single sample. In this case, the number of syndromes at our disposal is simply determined by the number (N) of individual classifiers in the combination we are working with, exemplified in Equation 17. In our work with malicious web scraping detectors, we are dealing with only two classifiers, meaning that our total possible number of syndromes is limited to four.

$$Syndromes = 2^N \tag{17}$$

For our study with recurrent neural networks however, this is not the case, as the outputs generated by the individual classifiers is not binary, but rather a continuous value between 0 (benign) and 1 (malicious). Table 37 is actually a real combination of RNN models from our study in chapter 5. By establishing a threshold point, this becomes analogous to a binary system - for example, with a threshold of 0.5 the classifiers either generate a prediction between 0 and 0.5 (benign) or between 0.5 and 1 (malicious). For simplicity in our work, we refer to these benign and malicious ranges as we would with a binary system - by labelling them as 0 (benign) or 1 (malicious). However, non-binary systems can be further expanded. Rather than dictating that predictions generated should fall between two discrete intervals, we could decide to make three discrete intervals (such as 0 to 0.33, 0.33 to 0.66 and 0.66 to 1), or more. In these non-binary cases, the number of possible syndromes is dependent not only on the number of classifiers, but also on the number of thresholds (T) we define, exemplified in 18. The higher the number of threshold points that dictate these intervals, the more individual syndromes are available for a system to decide it's overall prediction. This insight is a crucial advantage that non-binary systems have when applying optimal adjudication, over binary classification systems.

$$Syndromes = (T + 1)^N \tag{18}$$

An additional aspect of an optimal adjudication scheme is that it can take into account different values for the cost incurred from generating false positives and false negatives. In a conventional adjudication scheme, for example in a 1-out-of-N, the number of false positives and false negatives generated is simply dependent on the outputs of the individual classifiers. This is not an issue if the outputs generated by the different classifiers can be fine-tuned prior to their deployment in order to balance out the number of false positives and false negatives that are generated overall. However, if the costs incurred from these errors are changed, then recalibrating the individual classifiers would be required to rebalance these numbers again.

With an optimal adjudication, these changes are more resilient. By looking at the same historical data for when different syndromes are generated, the value of a false negative and false positive can be easily incorporated into the adjudication decision by simply multiplying the number of positives and negatives that generated that syndrome. In this way an optimal adjudication scheme is a more adaptable to changes and requirements of organisations as they change over time.

6.2 Web scrapers optimal adjudication

6.2.1 Introduction

In this section we will use the dataset we have introduced in our previous chapter 4 to exemplify the construction of an optimal adjudicator. Because in these datasets we only have access to two classifiers, both of which are strictly binary classifiers, the usefulness of an optimal adjudication is rather limited, as we described previously. Being limited to only four possible syndromes severely limits the results that an optimal adjudicator can achieve. It is rare that an optimal adjudicator with only four possible syndromes functions differently to other conventional adjudication schemes we're presented previously (i.e., 1oo2 and 2oo2). The only other two options are optimal adjudicators that never alert or always alert, or optimal adjudicators that strictly follow the classifications given by one of its internal classifiers. Nonetheless, optimal adjudication still offers one benefit in these cases, that being the ability to adapt to changing error costs. By being able to stipulate how much a false positive and a false negative cost, one is able to adapt the optimal adjudicator to switch between these different "modes of function" such as to minimise total error costs.

Because of the limited nature of these datasets, we have also opted to exclude the training and testing aspect of the creation of optimal adjudicators. Because optimal adjudication is based on historical data, namely, the number of positive and negative samples previously observed for each syndrome, in order to properly access the performance of an optimal adjudicator, one should divide a sample dataset into training and testing, much in the same way one would construct a machine learning model. This is because, when "training" an optimal adjudicator, one cannot be confident that the data used for training accurately represents what types of samples one would encounter in a production environment. Because we have chosen to use this limited dataset as an example of optimal adjudication, we have opted to use the entire dataset for both "training" and "testing", meaning that essentially, we are assuming our training dataset perfectly represents what the classifier would encounter in a production environment.

Because this is an extension into our work presented in chapter 4 we will not be detailing any of the tools or the datasets this is based upon.

6.2.2 Results

First, let us identify the adjudication outputs in regard to each dataset. In total we have four syndromes to work with, as follows: none of the tools alert, only Arcane alerts, only CommTool alerts or both tools alert. We represent the adjudication outputs in Table 38 for all three datasets (D1, D2 and D3), where a black cell represents a syndrome where an alert would be raised by the optimal adjudicator, while an empty cell represents a syndrome where an alert would not be raised. In all three datasets, we are dealing with bot-heavy cases. For D1, this means that an optimal adjudicator always raises an alert, regardless of the syndrome - this includes the situation when neither Arcane nor CommTool raise an alert themselves. In D2 and D3, we see the optimal adjudicator act as a 1oo2 system, raising an alert if either Arcane or CommTool raise an alert.

As expected, a mere four syndromes do not allow an optimal adjudicator to excel past more conventional schemes. However, as we discussed previously, the ability for an optimal adjudicator to take into account error costs is still present in these more trivial situations. In Table 38 we are assuming a 1 to 1 value for the losses incurred from alerting or not alerting. This means that we have determined that false positives and false negatives have

Table 38: Optimal Adjudication Output. White cell: no alert; Black cell: alert

	None	**Arcane only**	**CommTool only**	**Both**
D1	■	■	■	■
D2		■	■	■
D3		■	■	■

Table 39: Optimal Adjudication Output with Varying FP and FN Costs for D1. White cell: no alert; Black cell: alert

FP:FN ratio	**None**	**Arcane only**	**CommTool only**	**Both**
100:1 to 33:1	□	□	■	■
32:1 to 12:1	□	■	■	■
11:1 to 1:100	■	■	■	■

Table 40: Optimal Adjudication Output with Varying FP and FN Costs for D2. White cell: no alert; Black cell: alert

FP:FN ratio	**None**	**Arcane only**	**CommTool only**	**Both**
100:1 to 83:1	□	□	□	■
82:1 to 16:1	□	■	□	■
15:1 to 1:3	□	■	■	■
1:4 to 1:100	■	■	■	■

Table 41: Optimal Adjudication Output with Varying FP and FN Costs for D3. White cell: no alert; Black cell: alert

FP:FN ratio	**None**	**Arcane only**	**CommTool only**	**Both**
100:1 to 1:3	□	■	■	■
1:4 to 1:100	■	■	■	■

the same cost associated with them, and the output of our optimal adjudicator is simply based on whether there are more positive or negative samples on a per-syndrome basis. This 1 to 1 cost ratio is uncommon in security scenarios. In our context of malicious web scraping for example, a false positive is more costly than a false negative, because the impact of a single bot session is relatively minimal compared to blocking a legitimate user from making a purchase. In Tables 39, 40 and 41 we show how the optimal adjudicator changes if we change the costs associated with false positives and false negatives for all three datasets, respectively. For example, we can see in Table 39 that, when valuing false positives against false negatives between 100 to 1 and 33 to 1, an optimal adjudication scheme alerts if either CommTool alerts, or both CommTool and Arcane alerts, but never when only Arcane alerts. This changes when valuing false positives between 32 to 1 and 12 to 1 against false negatives, where the optimal adjudicator would act as a 1oo2 system, raising an alert if either Arcane and/or CommTool alert.

We can thus compare the total error costs achieved by these optimal adjudicators to the total error costs achieved by conventional adjudication schemes. We do this in Figs. 32, 33 and 34 where we show the total losses incurred, on the y-axis, as we change the FP:FN cost ratio along the x-axis. We show this for an optimal adjudicator, as well as for Arcane, CommTool, 1oo2 and 2oo2. We can see that the optimal adjudicator always incurs the least amount of losses, which is the expected behaviour. But this analysis allows us to quantify how much better would an optimal adjudicator perform, for different cost of failure scenarios.

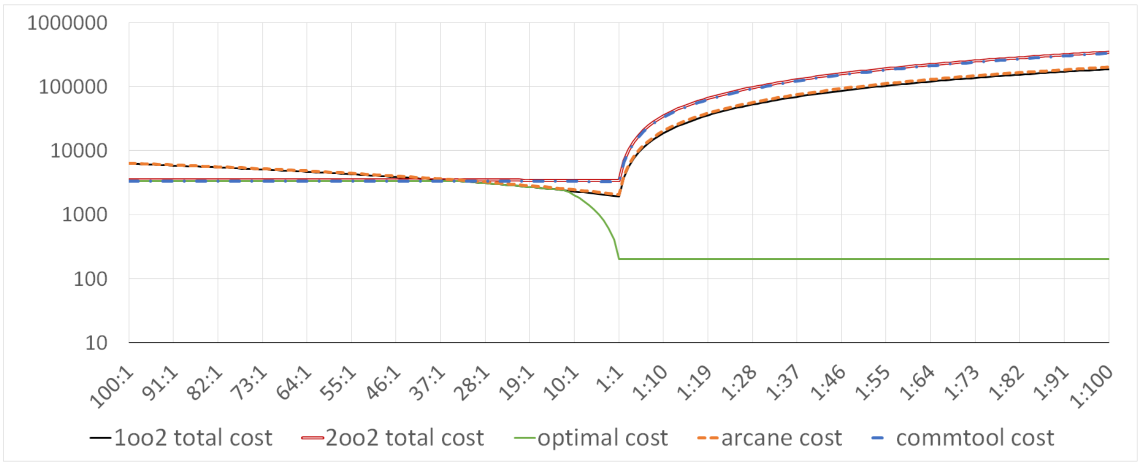


Figure 32: D1 total error costs

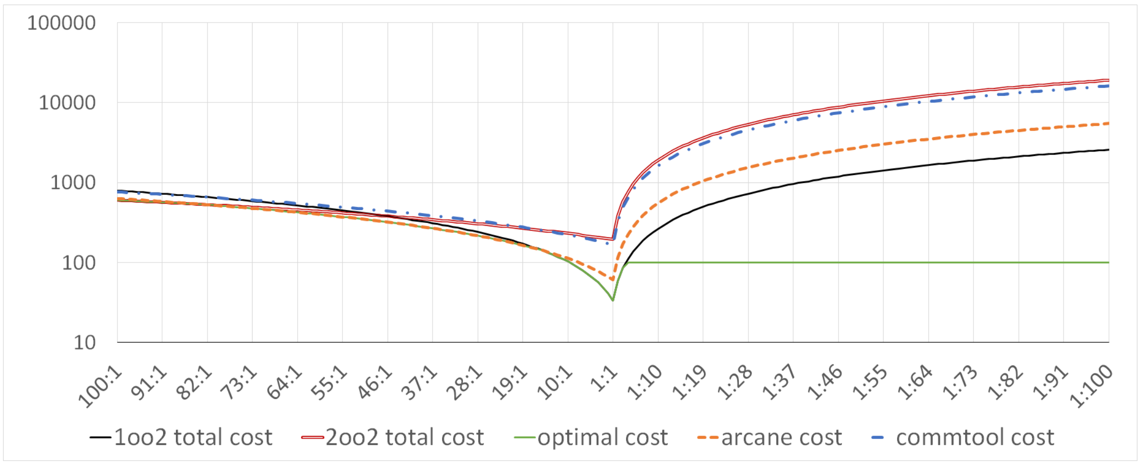


Figure 33: D2 total error costs

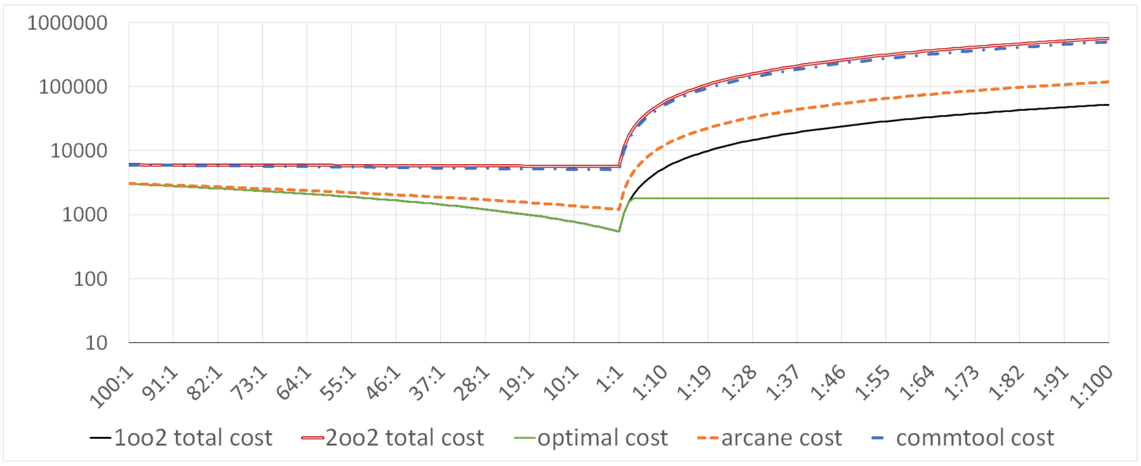


Figure 34: D3 total error costs

6.3 RNN optimal adjudication

6.3.1 Introduction

In this section we take a look at the application of optimal adjudication to our study with RNN machine learning models for software classification, which we introduced in chapter 5. While our previous application of optimal adjudication in the context of malicious web scraping was rather limited in the benefits it could achieve, our opportunity for performance benefits with this RNN dataset is much greater. This is for two reasons: i) the fact that we have 37 models to create combinations out of, instead of only two; ii) the fact that the output of each model is a value between 0 and 1 rather than binary outputs. This leads to the creation of a wide range of syndromes for the various combinations of models, allowing for the creation of adjudication schemes which are significantly different from conventional ones.

Our analysis is this section can be split into two parts. The first part deals with the comparison of basic optimal adjudication against the conventional adjudication schemes we've presented before for this dataset (chapter 5). To do this, we've created the optimal adjudicators for all possible combinations of 3 and 5 models, from our initial pool of 37 RNN models. We directly compare these against the metrics obtained by the conventional adjudication schemes we've calculated previously. While previously we calculate combinations of up to 9 models, we've limited it here to combinations of 3 and 5, due to the increased computational complexity that comes with constructing optimal adjudicators. The second part of our analysis concerns the inherent advantages that optimal adjudicators have themselves, in their construction and in their mode of operation, with a more in-depth look at the benefits afforded from the ability to balance the cost of false positives and false negatives.

Because this is an extension into our work presented in chapter 5 we will not be detailing any of the tools or the dataset this is based upon.

6.3.2 Methodology

Of the total pool of 37 RNN models present in the dataset first introduced in chapter 5, we've constructed all possible optimal adjudicators in combinations of 3 and 5 models, in order to create a direct comparison with the conventional adjudication schemes we presented before. We've chosen combinations of 3 and 5 specifically, so as to be able to compare optimal adjudicators with simple majority schemes, as these are the most similar - a simple majority scheme maximises accuracy, while an optimal adjudicator with equal cost false positives and false negatives does the same.

A significant difference between this study and our previous application of optimal adjudication deals with the fact that the classifiers used in this dataset (the RNN models that make up an optimal adjudicator) are not binary classifiers, but rather generate predictions between 0 (benign) and 1 (malicious), with a prediction of 0.5 or above being considered malicious under normal conditions. In order to define the syndromes of each optimal adjudicator we need to define ranges of intervals that dictate what each prediction generated by a classifier represents. The simplest version of this is to dictate a single threshold point of 0.5, where any prediction below that is classified as benign, and any prediction above is classified as malicious. When constructing our optimal adjudicators in combinations of both 3 and 5 models, we've used this single threshold value of 0.5. In our next sections we will refer to these optimal adjudicators with a single threshold point as "Optimal-T1". However, the use of additional thresholds points when creating an optimal adjudicator leads to the creation of additional syndromes, which can lead to a more accurate system. For combinations of 3 models, we've additionally calculated optimal adjudicators that use 2, 3 and 4 threshold points (creating equally sized ranges of potential predictions). We refer to these optimal adjudicators as "Optimal-T2", "Optimal-T3" and "Optimal-T4".

One aspect which we did not accurately represent in our previous application of optimal adjudication is the need to use a training and testing methodology when benchmarking the performance of our systems. Much like the construction of a machine learning model, a separation of training data and testing data is essential in order to have confidence in the performance results of a system. In this work we've done this by splitting our dataset into thirds. Two thirds were used for training, while the remaining one third used for testing. When comparing our results with those obtained with conventional adjudication schemes we've gone back and calculated performance achieved by the conventional adjudicators for only that same one third used for testing, such that our optimal adjudicators are directly comparable with our previous results.

The defining feature of an optimal adjudicator is that it minimises the total error cost obtained when generating false positives and false negatives. In the case where a false positive and a false negative are both valued the same, the total error cost of the system will simply be the total number of false classifications generated, multiplied by the cost of a single misclassification. In these cases, this will have the same effect as maximising accuracy. However, if we attribute different error costs to different error types, then the accuracy of a system can be decreased such as to minimise the total error cost of said system. We can reason about the total error cost of a system without using an absolute cost, by looking instead at the mean error cost that each demand presents. When classifying a sample, an optimal adjudicator can generate a correct prediction, in which case the cost of that demand will be 0 (no error cost), or it can generate either a false positive or false negative, in which case the cost of that demand will be either the cost of a false positive (L_{fp}) or the cost of a false negative (L_{fn}). The value of the expected loss of a demand can thus be define as

$$E[L] = (L_{fp}P_{fp} + L_{fn}P_{fn}) \tag{19}$$

where P_{fp} denotes the probability of a false positive and P_{fn} the probability of a false negative. In our previous application of optimal adjudication, in section 6.2 we defined predetermined error cost ratios between false positives and false negatives (such as 1 to 1, 10 to 1, etc). A better way of defining these cost ratios is instead to combine both error costs into a single variable $\gamma = \frac{L_{fp}}{L_{fn}}$, which we can then use to plot our expected loss in a discrete manner along a single axis. If we assign a fixed cost of 1 to our false negative classifications, this means that our γ variable is simply equal to the cost associated with a false positive (L_{fp}), and we can thus rewrite our formula for the expected loss of a demand as:

$$E[L] = \gamma P_{fp} + P_{fn} \tag{20}$$

We make use of this variable γ prominently in our analysis, where values of γ between 0 and 1 represent cases where false positives induce a smaller error cost compared to false negatives, and values greater than 1 represent cases where false positives induce a greater error cost compared to false negatives.

6.3.3 Results

6.3.3.1 Optimal adjudication vs. conventional schemes

Let us start by directly comparing the performance of optimal adjudicators against the performance achieved by conventional adjudication schemes, without the introduction of different error costs. In a 1 to 1 cost ratio, an optimal adjudicator is a direct improvement of simple majority schemes, as both are built to maximise accuracy. In Fig. 35 we compare the accuracy achieved by optimal adjudicators compared to the simple majority schemes we presented in chapter 5, for combinations of 3 and 5 models[15].

Directly apparent is the increase in accuracy achieved by optimal adjudicators compared to simple majority schemes. For combinations of 3 models, the median accuracy for an optimal adjudicator is 0.71 compared to the simple majority which achieved 0.69. This increase becomes more prominent for combinations of 5 models, where optimal adjudicators achieve a median accuracy of 0.76, while simple majority schemes continue to show a median accuracy of 0.69. Recall back to the analysis presented in chapter 5, where simple majority schemes did not increase the median accuracy observed for a system as the number of models in that system increased, even in combinations of 7 and 9 models.

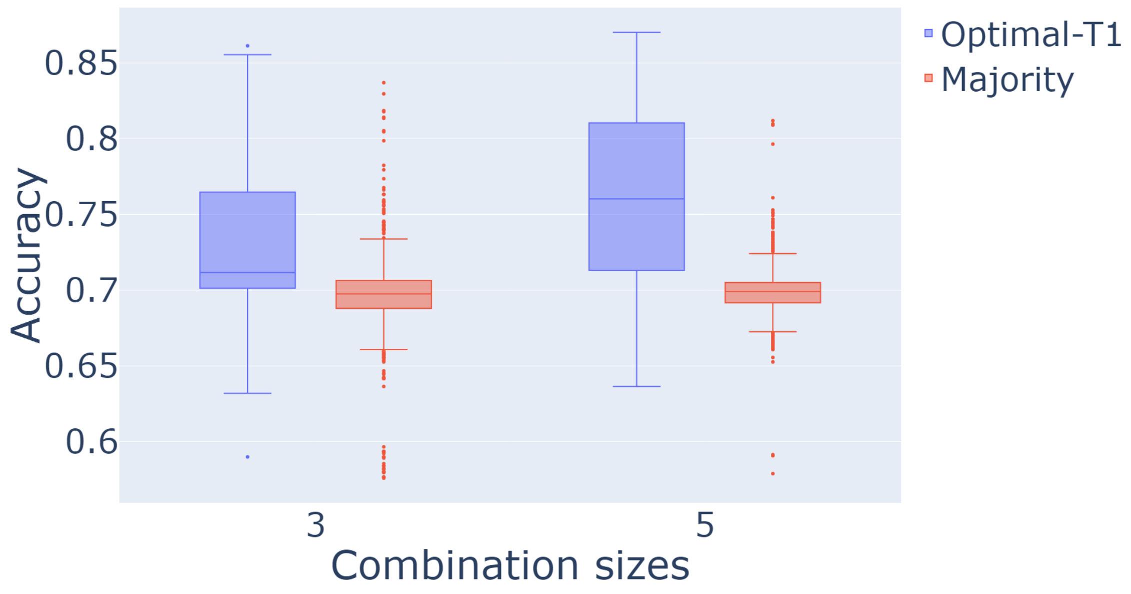


Figure 35: Accuracy for optimal adjudicators T1 and simple majority schemes

[15]Again, we've gone back and used only the testing partition of the dataset to benchmark the previous conventional adjudication schemes.

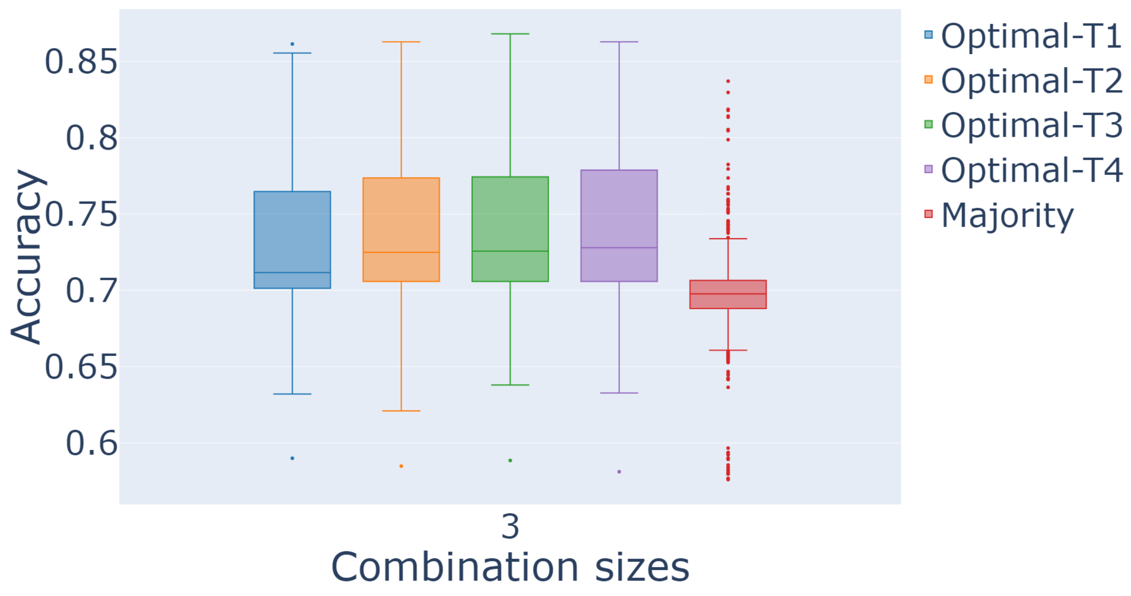


Figure 36: Accuracy for optimal adjudicators T1, T2 and T4

Here however, with optimal adjudicators, the increased number of models per combination does lead to a significant positive difference in performance, in terms of overall accuracy.

We can further improve the accuracy seen in optimal adjudicators by increasing the number of thresholds when building our systems, as we explained in our methodology. In Fig. 36 we show how this increases in combinations of 3 models, when constructing optimal adjudicators with 2, 3 and 4 threshold points. When creating an optimal adjudicator with the use of 2 threshold points, we can increase the median accuracy achieved from 0.71 to 0.72. Adding additional thresholds past this point however has little effect on the observed accuracy performance of the optimal adjudicator.

6.3.3.2 Optimal adjudication with varying error costs

We've seen now how, when dealing with cost ratios of 1 to 1 in terms of false positives and false negatives, optimal adjudicators achieve the highest possible accuracy of any adjudication scheme shown so far. This however, is simply a by-product of the optimal adjudicators actual function, which is to minimise total error costs. When dealing with any situation where the cost of false positives and false negatives are different, the optimal adjudicator will forfeit some amount of overall accuracy, in order to decrease the total error cost observed by the system. We can observe the effects that different cost ratios have on an optimal adjudicator by using our γ variable we introduced in our methodology and plotting the mean cost per demand as γ changes. We begin by exploring only a single instance of an optimal adjudicator, in Fig. 37, which represents the cost per demand profile achieved by an optimal adjudicator made up of RNN models 10, 12 and 23, with a single threshold point of 0.5.

You will notice that there are 8 pairs of vertical markers that make up the line of our optimal adjudicator. These marker pairs correspond to the 8 syndromes that make up the adjudicator, and to the points where each syndrome's response switched from ”alerting”

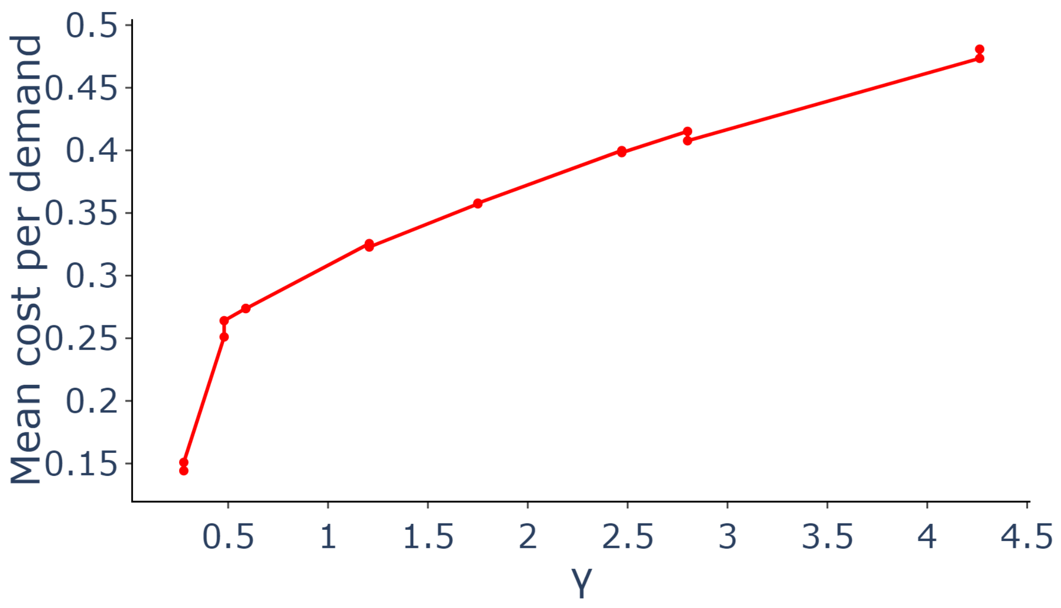


Figure 37: Mean cost per demand profile for a single optimal adjudicator (RNN models 10, 12 and 23)

to "not alerting", as the value of γ changes, reading from left to right. The left-most marker pair, closest to the point $\gamma = 0$, represents the point at which all but one of the syndromes' responses were to "alert". Each marker pair after this represents another of those 8 syndromes' responses switching from "alert" to "not alert", until the right most marker pair where all 8 syndromes' responses were "not alert". The right-most point represents the maximum cost per demand possible for this optimal adjudicator, with any further γ increases resulting in no additional increases to the mean error cost. Any point to the left of the first syndrome marker pair will incur a linear decrease whose slope is defined by the probability of a false positive (P_{fp}), until the mean cost per demand is equal to zero.

Between each of the markers in a pair, our line can either move upwards or downwards (or stay the same). The reason for this is due to the data used for training not accurately representing the validation data used for testing. Based on the training data, an optimal adjudicator might, for example, decide to switch one of its syndromes from "alert" to "not alert", at the point $\gamma = 2.8$, as is roughly the case in our figure. This would be the point at which, based on the training data used the cost of false positives outweighted the cost of false negatives for that syndrome. However, for the testing data which we are using to validate our system, that point at which false positives outweighted the cost of false negatives actually occurred at an earlier point of γ, hence our mean error cost jumps down vertically, due to our optimal adjudication switching its output "too late". Similarly, our optimal adjudicator might switch its output "too early", leading to a vertical increase at one of these marker pairs. In the case where our training data accurately represents the testing data all of these markers would coincide vertically. We can observe the increased error cost from this inaccuracy in training data if we create the same optimal adjudicator but use the testing data as both testing and training, as we have done in Fig. 38, where the line "Unknown P-rate" represents the optimal adjudicator built with an inaccurate

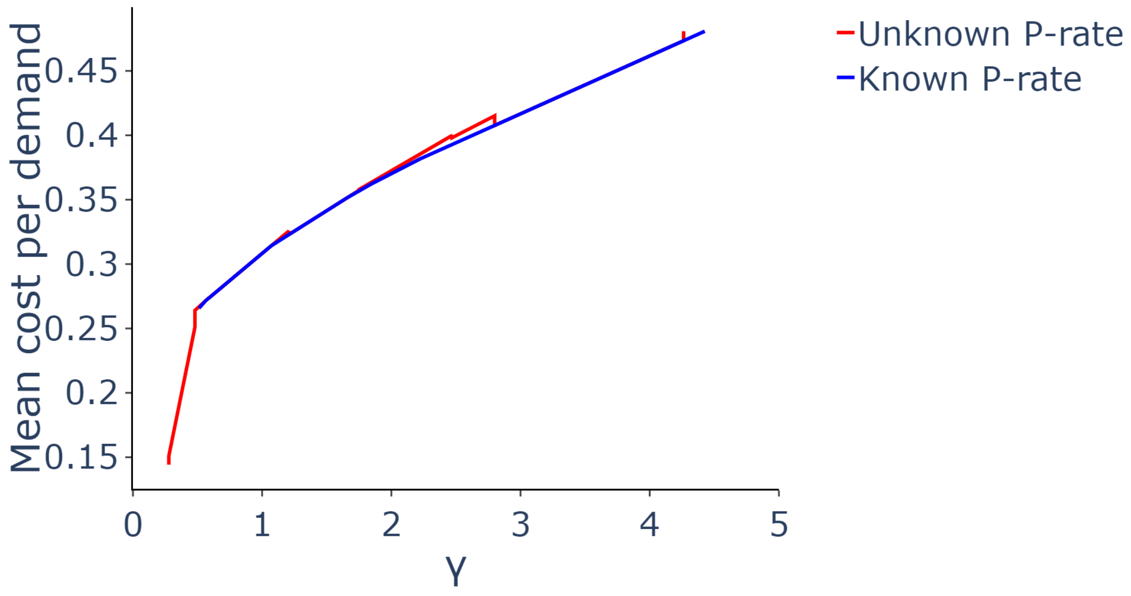


Figure 38: Mean cost per demand profile for a single optimal adjudicator - Known vs Unknown P-rates (RNN models 10, 12 and 23)

training dataset (hence we don't know the true values of the probability of positive and negative samples), and the line "Known P-rate" represents the optimal adjudicator built using the testing portion of the dataset for both training and validation[16].

The mean error cost of an optimal adjudicator built with inaccurate training data is always equal to, or greater than the cost of a completely accurately trained adjudicator. The mean cost at any point of the diagonal segments that connect different γ marker pairs will simply be a value between the costs at either end, described by the linear increase in γ. One aspect to note is that there is equal amounts of information on either side of the point $\gamma = 1$. We can see this if we instead use a logarithmic function for our x-axis, as shown in Fig. 39.

We can create the same plots comparing our optimal adjudicators with the performance achieved by conventional adjudication schemes. We show this in Figs. 40 and 41 where we plot an average of all optimal adjudicators against the average performances of 1ooN, NooN and simple majority schemes, in combinations of 3 and 5 models, respectively[17].

The most important aspect to consider of these plots is the different scales at which the various adjudication schemes increase in mean error cost per demand. Conventional schemes (1ooN, NooN and simple majority) do not take into account error costs when generating their predictions, and as such they show a linear increase in mean cost as these cost ratios change. In contrast, optimal adjudicators do not increase linearly as we saw previously. We want to place a special emphasis on the small portion of Fig. 40 where the mean error cost for an optimal adjudicator actually rises above that observed for a

[16]Our "Unknown P-rate" line may mislead a reader into thinking it achieves a lower mean cost per demand at lower values of γ. This is a result of this optimal adjudicator making its first two syndrome changes "too early", hence the two left-most vertical jumps on this line. Since we are only plotting these lines from the point at which all syndromes generate an "alert" output, the linear decreases that would show the "Known P-rate" line as the lowest mean cost are not depicted.

[17]Fig. 41 represents an average of 20 classifiers rather than all. This is to allows us to show the same amount of detail while saving computational resources.

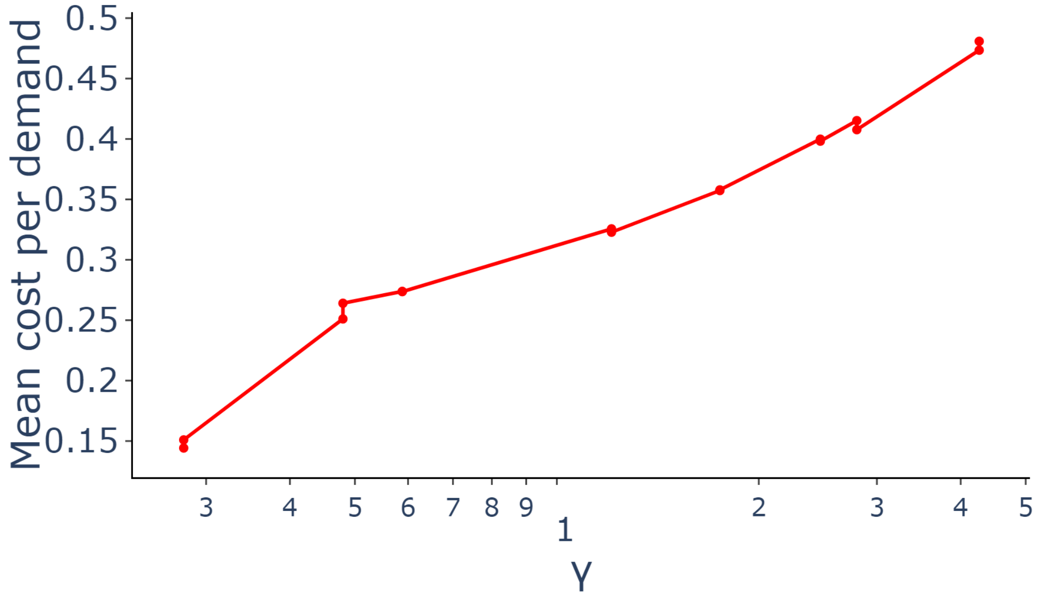


Figure 39: Mean cost per demand profile for a single optimal adjudicator (log)

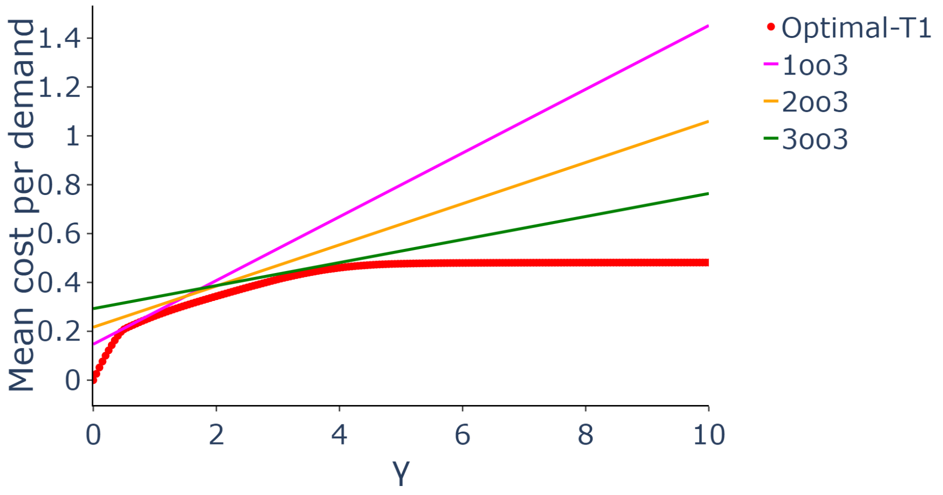


Figure 40: Mean cost per demand for optimal adjudicator T1, 1oo3, 2oo3 and 3oo3 (n=3)

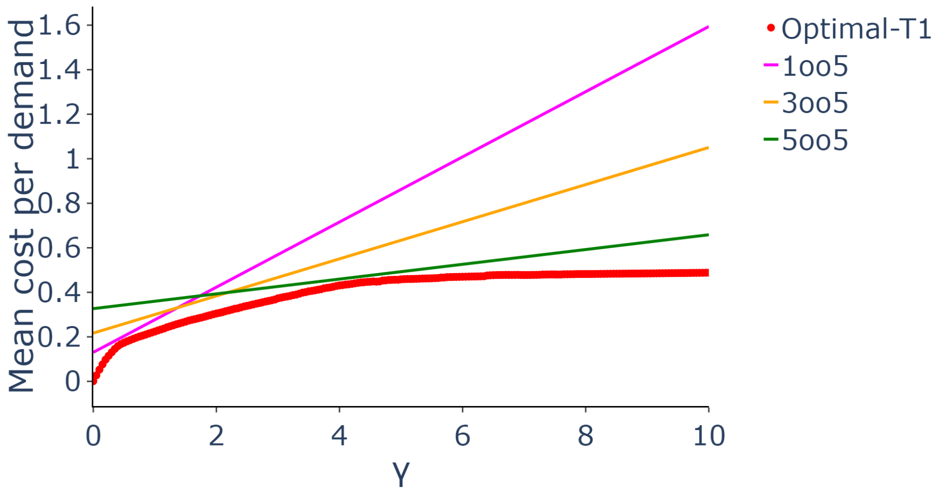


Figure 41: Mean cost per demand for optimal adjudicator T1, 1oo5, 3oo5 and 5oo5 (n=5)

1oo3 adjudication scheme, roughly at $\gamma = 0.5$. This is counter intuitive and seems to be a contradiction of what we stated previously, which is that optimal adjudicators always achieve the least error cost total compared to conventional adjudication schemes. The reason that this can happen is due to the problem we've highlighted previously, which is in the selection of data used for training an optimal adjudicator. In some cases, the discrepancies we saw in Fig. 38 between optimal adjudicators built with accurate and inaccurate datasets will also be present when comparing optimal adjudicators against conventional adjudication schemes.

6.3.3.3 Thresholds for non-binary classification

As we did previously looking at the accuracy of optimal adjudicators built using different numbers of threshold points, let us do the same for our mean cost analysis. In Fig. 42 we show this for optimal adjudicators built with 1, 2, 3 and 4 threshold points, using an average of 20 classifiers of each class[18].

For the most part, optimal adjudicators built with more threshold points achieve a lower mean cost per demand. However, this is not the case on the extreme points, where the order actually inverts, with optimal adjudicators which were built with more threshold points achieving a higher mean cost per demand.

Under the assumption of accurate datasets, the additional syndromes created from the use of more threshold points would achieve an equal or lower cost, because each smaller syndrome becomes more accurate. We can observe this if we reconstruct the optimal adjudicators using the testing partition of the dataset for both training and testing, as shown in Fig. 43. We don't see these results in optimal adjudicators trained and tested on different datasets due to the inaccuracy inherent to the training datasets coupled with a "dilution" of each syndrome. By creating smaller syndromes, each syndrome has less data to make a decision with, and in many cases, we begin to run into problems of having no data at all. In these cases, the decision to alert or not alert is a difficult one. For example, if a particular syndrome never observes any samples (whether positive or negative) in our training dataset, we might decide to not alert by default, in which case we might be generating unnecessary false negatives if in our validation data we see more positive samples than negative samples. Similarly, we might have cases where a syndrome sees positive samples during training, but never any negative samples. In these cases that particular syndrome would always alert based solely on its training, but obviously this means that, if our validation data does contain negative samples, that optimal adjudicator will continuously incur additional costs as the value of γ increases.

These issues of lacking data, where particular syndromes might never have been triggered by a positive or negative sample (or both) is one that can also happen in optimal adjudicators built with a single threshold. However, this is not as common in our analysis because the amount of training data we have at our disposal is enough to populate eight syndromes per optimal adjudicator. As we add additional thresholds, this is not the case, and we begin to see more and more syndromes that have no training data to reasonably

[18]We've used only 20 classifiers with the objective of showing as much detail as possible without increasing computational complexity. We've used the same 20 combinations of RNN models for each class of classifiers, meaning these are directly comparable.

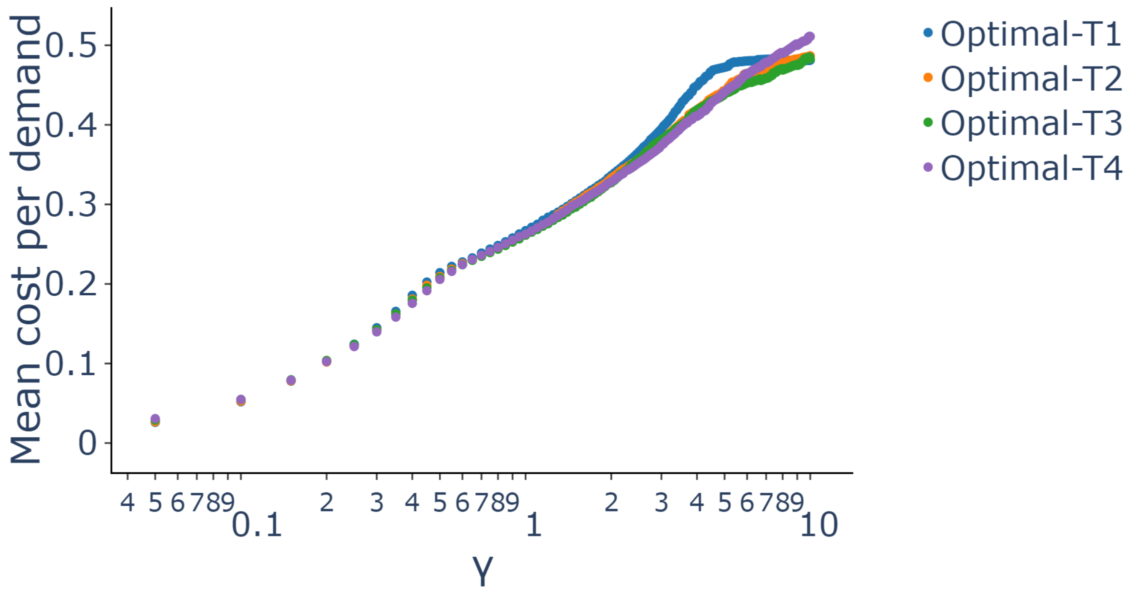


Figure 42: Mean cost per demand profile for optimal adjudicators T1, T2, T3 and T4 (log)

back up either decision.

There is another issue that can affect how optimal adjudicators built with different amounts of threshold points compare with each other, and this has to do with how threshold points are selected. For this, take a look at Fig. 44, where we show some possibilities of how to define threshold points.

Here we represent the space of all possible outputs of a single classifier by a vertical bar, with the output of 0 (benign) at the bottom, and the output of 1 (malicious) at the top. When defining threshold points, we can place them anywhere along this range of values. For optimal adjudicators making use of a single threshold point, it makes sense to place it at the value 0.5, as this is the natural threshold of classification for the machine learning models themselves. When defining two threshold points, we are now presented with an important choice. We can either define two brand new threshold points, different to the value used when defining a single threshold (represented by the bar T2), or we can maintain previously defined thresholds and simply add new ones in between the already existing ones (represented by the bar T2*). The important distinction here is that, by continuing with already existing threshold points, we are able to cleanly and completely represent any syndromes from previous threshold amounts by combining some number of the newly created syndromes. For example, in Fig. 44, the lower syndrome seen in T1 is made up of a combination of the two lowest syndromes in T2*, while the top syndrome in T1 is simply the same in T2*. This is not the case when existing threshold points are not maintained, meaning we cannot make these claims about T2.

We can leverage this idea to mitigate some of the issues that arise from lacking data in each syndrome, by building two or more, ”threshold-compatible” optimal adjudicators (by which we mean that each higher number threshold optimal adjudicator maintains the threshold points defined by previous optimal adjudicators). In these cases, the additional granularity gained from higher amounts of syndromes can be leveraged if these saw enough

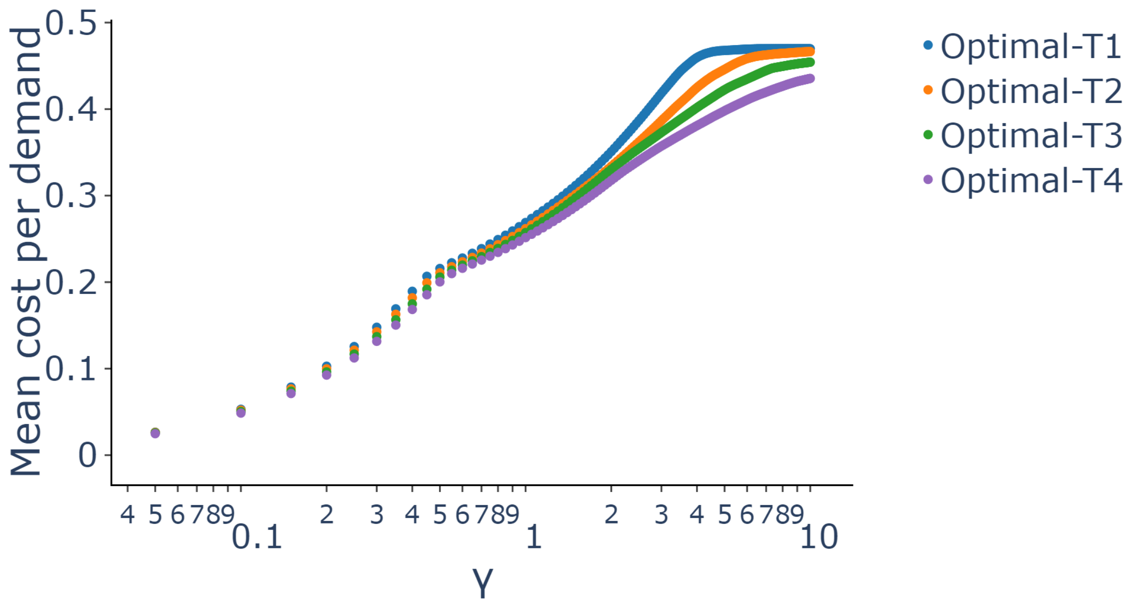


Figure 43: Mean cost per demand profile for optimal adjudicators T1, T2, T3 and T4 using testing data only (log)

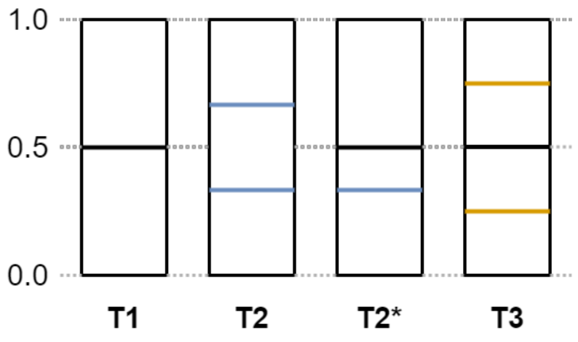


Figure 44: Different possibilities when defining thresholds

data during training to make a decision, while if they don't have data to indicate either classification, the decision can be delegated to the syndrome it composes in the equivalent, lower number threshold optimal adjudicator. This way, a higher threshold optimal adjudicator can be guaranteed to do no worse than its lower threshold counterpart.

In our analysis, we've defined thresholds such that they evenly divide the range of possible outputs of a classifier. This means that we actually have two "threshold-compatible" optimal adjudicators, represented in our diagram as T1 and T3. However, in our analysis we did not implement the logic of delegating decisions to lower threshold optimal adjudicators when not enough data is seen during training, and so our T3 optimal adjudicator does in fact achieve a higher cost at the extremes of our γ plots.

6.3.3.4 Sources of uncertainty and selecting the right γ

When constructing an optimal adjudicator, there are two sources of uncertainty that one has to deal with. We've talked already about the first of these sources, dealing with the uncertainty in the training datasets. In essence, this uncertainty is the uncertainty about the rates of samples in production, how many positives and negatives one should expect to encounter, and how these will map to the syndromes defined by an optimal adjudicator.

The second source of uncertainty stems from trying to accurately determine the costs of different types of failures (false positives and false negatives). So far, we've essentially varied our values of γ and calculated the mean cost per demand an optimal adjudicator would achieve under that value. However, in a production environment, there will be a single correct value of γ at any one time, one which accurately represents the cost ratio between false positives and false negatives. Determining what that value is, is a difficult task, and one which an assessor will often answer with a range of possible γ values rather than a single point. In doing so, it's important to assess how much worse an optimal adjudicator would be if an incorrect value is used for γ.

If training inaccuracies are present in an optimal adjudicator, the cost of that adjudicator will necessarily be greater for at least one point along the γ axis, as at least one of its syndromes will choose to "switch" it's output before or after that point. With an uncertainty about the correct value of γ also present however, both of these inaccuracies can cancel each other out. Let's exemplify this by comparing the performance of two optimal adjudicators, one built with the normal training and testing split in data, and the other both trained and tested with the same testing partition. Fig. 45 shows the example of RNN models 30, 32 and 33 in our dataset.

What we're interested here is determining where both lines have the same mean cost per demand. If the lines overlap each other, then this means that both optimal adjudicators achieve the same cost on the same value of γ, meaning that inaccuracies we may have in training data are not sufficient to trigger different outputs on a syndrome basis. If however, our lines achieve equal cost when both optimal adjudicator lines do not overlap, then we have a case where there are inaccuracies in our training data, but these inaccuracies are countered by a similar inaccuracy in our chosen γ value - because both optimal adjudicators achieve the same mean cost using different values of γ.

We can more easily analyse what this means by plotting pairs of γ values that lead to equal mean costs in both optimal adjudicators, which we've done in Fig. 46, where we plot along the x-axis values of γ associated with a completely accurate optimal adjudicator, and along the y-axis values of γ associated with our inaccurately trained optimal adjudicator.

For the most part in this example, this results in a diagonal line, meaning that for most values of γ the impact of training inaccuracies is not felt (because both optimal adjudicators achieve the same mean cost using the same value of γ). However, we do encounter areas where ranges of γ for one line map to differently sized ranges of γ for the other. This is most notable in our example, between the values of 5 and 7 for the x-axis, where a section of our equal cost line shows a less than 45-degree angle. We can expand

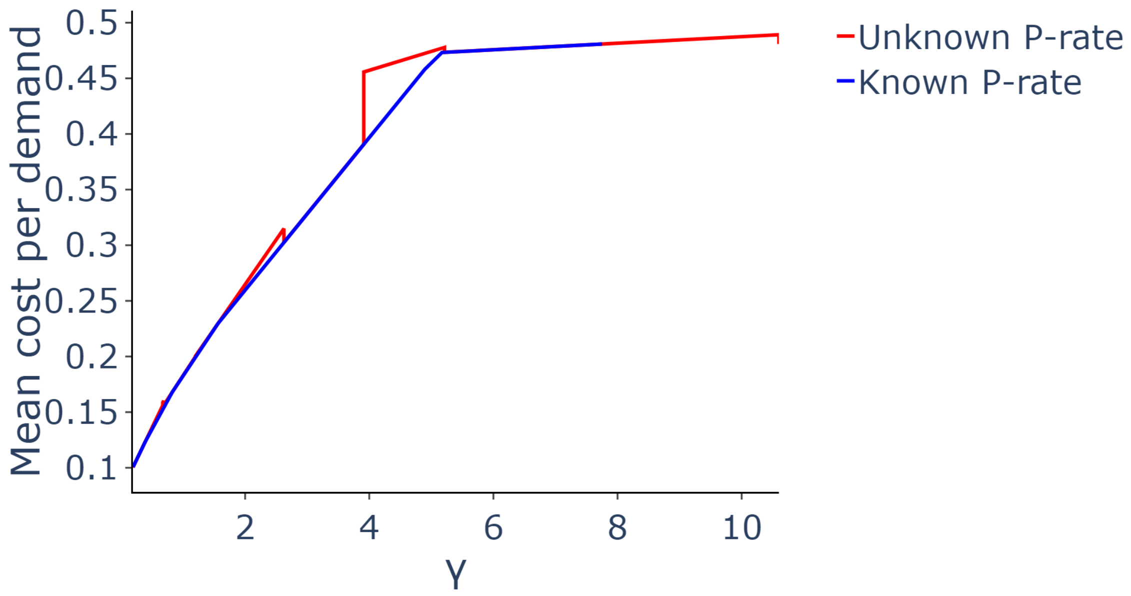


Figure 45: Mean cost per demand profile (RNN models 30, 32 and 33)

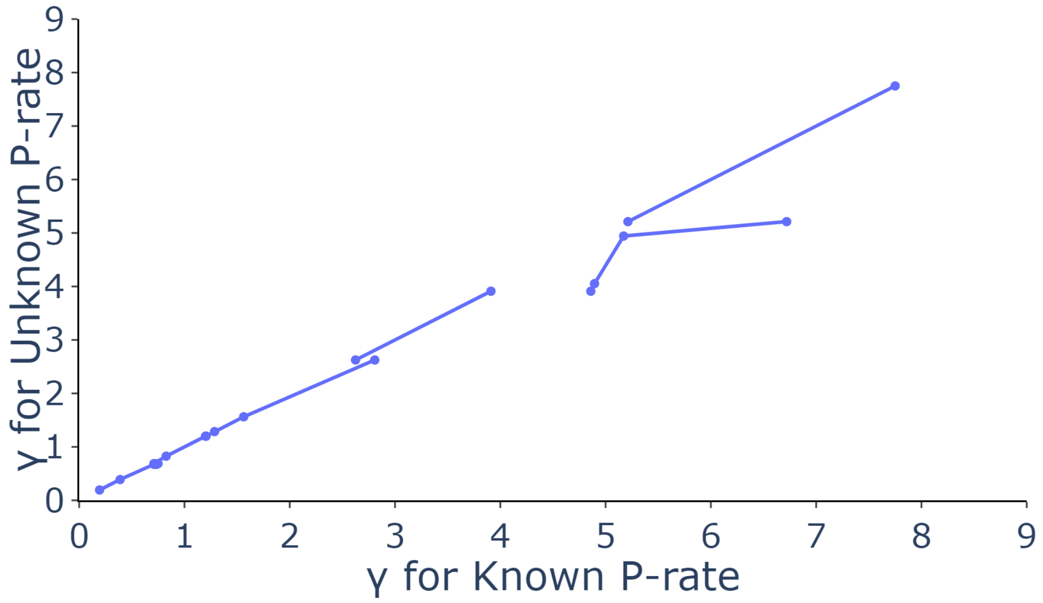


Figure 46: Ranges of γ for known and unknown P-rate (RNN models 30, 32 and 33)

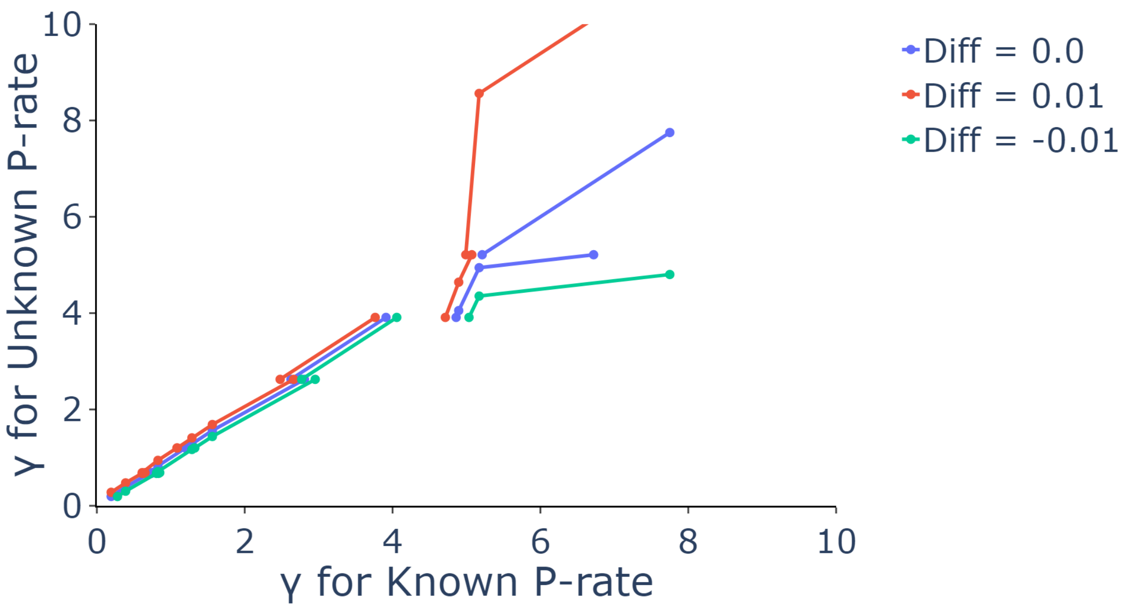


Figure 47: Ranges of γ for known and unknown P=rate, with different cost T cost differences (RNN models 30, 32 and 33)

this by, not only plotting the line of equal cost between the optimal adjudicators, but also lines of specifically defined differences in cost[19], such as 0.01 or -0.01, as we demonstrate in Fig. 47. These "bands" can be used to define confidence intervals, where certain γ values results in a cost difference of no more than X.

[19]Positive differences mean that our Unknown P-rate optimal adjudicator achieves a higher cost, while negative differences mean that our Known P-rate optimal adjudicator achieves a higher cost

Table 42: Positive and negative samples by syndrome (RNN models 30, 32 and 33)

Syndrome	**Training Positives**	**Training Negatives**	**Testing Positives**	**Testing Negatives**	γ **Unknown**	γ **Known**
A	212	1101	106	547	0.19	0.19
B	22	32	7	18	0.69	0.39
C	35	52	19	23	0.67	0.83
D	12	10	9	7	1.2	1.3
E	63	24	25	16	2.6	1.6
F	172	33	93	19	5.21	4.9
G	704	180	362	70	3.9	5.17
H	53	5	31	4	10.6	7.75

It may not yet be clear why training inaccuracies may only manifest themselves at certain points of γ. The reasoning behind this is that the entire output of an optimal adjudicator is determined on an individual syndrome basis and training inaccuracies may only affect a few syndromes of an optimal adjudicator. Take a look at Table 42 where we show the data that makes up each syndrome of the optimal adjudicators we've been looking at thus far (RNN models 30, 32 and 33). We show, for each of it's 8 syndromes: how many positives and negative samples were observed during training; how many positive and negative samples were observed during testing; and for both unknown and known P-rate lines, at which point of γ a particular syndrome switches from "alert" to "not alert", where unknown P-rate is going to be based solely on the training data, while known P-rate solely on the testing data.

The syndromes where known and unknown P-rate optimal adjudicators agree on a γ value are syndromes where our training data accurately represents the real rates of positives or negatives (or are very close to), such as syndromes A and D in our table. The syndromes where these γ don't coincident are syndromes where our training and testing datasets differ substantially (or have very little data). These differences in chosen γ values for each syndrome, and the sections of Figs. 46 and 47 where our different cost lines diverge from the diagonal essentially point us to syndromes where our training data presents inaccuracies. While we've not been able to find a real example from the dataset we have analysed, its reasonable to envision a case where all but one of our syndromes include perfectly accurate training data. In those cases, the mean cost of our optimal adjudicator, at the point of γ where that syndrome switches output, will still be affected the same whether or not our other syndromes have inaccurate training data.

6.4 Conclusions

In this chapter we have detailed our application of optimal adjudication techniques on the datasets we first introduced in chapters 4 and 5. We exemplify the construction of an optimal adjudicator with our datasets concerning the use of malicious web scraping detectors, but abstain from making substantial comments as the lack of possible syndromes that one could create with these datasets severely limits the benefits optimal adjudicators could achieve.

For our dataset concerning RNN machine learning models, we've employed a training and testing methodology to create optimal adjudicators, in combinations of 3 and 5 models, and compare these directly with the results we observed with the use of conventional adjudication schemes. In particular, we describe how the increase in the number of models per combination results in additional improvements to the accuracy of optimal adjudication schemes, while this same effect is not observed with simple majority schemes. We additionally detail these increases with the use of additional threshold points when constructing optimal adjudications.

As we delve deeper into the inherent advantages of the use of optimal adjudicators, we make a specific point of highlighting the necessity of correctly designing a training dataset that accurately represents the rates of positives and negatives an optimal adjudicator is going to be exposed to in a production environment. In some cases, these training inaccuracies can result in higher errors costs observed for an optimal adjudicator compared to a conventional adjudication scheme. This is something we advocate must be the focus of additional scrutiny when creating optimal adjudicators, as the creation of accurate training datasets is a difficult task itself. With this in mind, one may be interested in the results pursued in the fields of machine learning, where the creation of realistic training datasets represents a substantial effort [117], and similarly in the reliability field, in particular the definition of operational profiles [4]

We detail the process of determining threshold points when constructing optimal adjudicators with non-binary classifiers, and the complications that can arise from this, including the dilution of training data amongst smaller, more numerous syndromes, resulting in instances where some syndromes are never triggered by a sample during training and thus are unable to make informed decisions when such samples do occur in production. In doing so, we advocate for the integration of higher threshold optimal adjudicators "vertically" by utilising the output of the smallest possible syndrome when enough data was seen during training and delegating the output to lower threshold syndromes when this is not the case.

Furthermore, we describe the interplay that exists between the two sources of uncertain inherent to optimal adjudicators, being the uncertainty about the rates of positives and negatives, and the uncertainty when determining the correct cost associated with generating false positives and false negatives.

While our results pose interesting questions, one aspect of our methodology may lead to some bias. This is due to us having performed our study on a single training and testing split over our dataset. In other to have higher confidence in our results, our analysis should have been done over a large number of unique data splits, such as to minimise the risk of using a potentially skewed data split.

We are currently continuing work on extending the analysis of optimal adjudicators using our RNN machine learning models, in particularly looking at the confirmation and generalisation of the results seen here through analytical means.

7 Discussion and conclusions

In this work we have looked at the usage of design diversity in two security contexts. We first detailed the results obtained from analysing three different datasets regarding the detection of malicious web scraping activity, provided to us by an industrial partner, which contained the traffic observed within their network, including the alerts generated by two of their detection tools. Similarly, we analysed the potential of design diversity in regard to the use of RNN machine learning models to classify software samples as either benign or malicious. In both studies we detail the empirical results obtained from combining diverse tools for security, and then go on to highlight the sources of diversity between the tools. Finally, we present a comprehensive use of optimal adjudication in the context of design diversity, applying it to both contexts previously introduced, and motivate its usage for increased performance and reliability within the security context.

It is particularly important to generate empirical results for the use of design diversity in as many different contexts as possible. This is because, as Littlewood and Strigini point out [5], design diversity can yield significantly different benefits depending on the environment in which they are utilised. Anecdotally, this notion makes sense, as it is easy to envision how tackling different problems can differ significantly in the approaches one takes. Some problems have few possible solutions, meaning that there's not much to be gained from diversifying our approaches as compared to simply attempting to improve them individually. Alternatively, some problems may be so expansive as to allow for many different ways of solving them. In these cases, one can argue that applying diversity is an easy, and more successful approach.

Additionally, it is important that we pinpoint the best ways we can adequately quantify the improvement we can achieve with the use of design diversity. Different benchmarking metrics will be more or less useful in different contexts based on what our goal in that context is [3]. In this work we have focused on the system security landscape, and for this we have placed greater emphasis on the use of sensitivity and specificity. In security, both correctly detecting malicious samples, and minimising the number of false positives is important, and so both metrics are important in order to qualify the gains and drawbacks with the use of design diversity. While the cost of generating false positives or false negatives may differ, it's important to realise that these differences in cost will seldom be so extreme as to significantly overvalue one over the other, as is the case in other contexts, such as medical diagnosis.

In our first section of work, we have detailed the use of design diversity in the context of detecting malicious web scraping activity. We analysed three datasets containing the network traffic and alerts generated by two detection tools, Arcane and CommTool. These tools are quite different in the way they operate, with Arcane being exclusively server-side, while CommTool works mostly on the client-side. In theory, these are vastly different modes of operation, and in our analysis, we did observe, at times significantly, benefits on the use of both of these tools combined into one adjudication system. This was particularly relevant for the first dataset in question, where the combination of both tools in a 1-out-of-2 scheme increased the sensitivity of the system by up to 29.65 percentage points and a 2-out-of-2 scheme increased the specificity of the system by up to 21.32 points. However, after sharing the results of this first dataset analysis with the dataset provided, they reconfigured their tools, such that in the second dataset we analysed they had been improved. In this follow-up dataset analysis, we did not observe such a drastic improvement in sensitivity or specificity. While hardly any proof on the true benefits of design diversity in this context, we speculate that the significant increases we saw in our first dataset may not be due to the inherent differences between the two tools, but rather due to them simply not being as well configured as they could have been.

There are some examples we can point to that indicate diversity between the tools, an example being the alerting diversity based on the number of bytes sent per session. In all three datasets there was a tendency for Arcane to alert separately from CommTool for sessions with higher total number of bytes sent. While neither tool makes direct use of the number of bytes sent in their detection algorithm, this is still a noticeable difference in the tools' alerting patterns, and we posit that this is due to Arcane being server-side and thus generating more confident alerts the longer the sessions in question last.

Nonetheless, a vast majority of the diversity we observed in the alerting patterns of both tools we would categorise as diversity of methodology, without a diversity in outcomes. This meaning that both tools generally alert in conjunction for the same traffic samples, but do so through different means. A straightforward example of this is CommTool's use of JavaScript-based detection techniques. While Arcane does not utilise JavaScript at all, it nonetheless still alerted more often than not on the same particular samples for which CommTool alerted on based on JavaScript checks. Another example were the alerting patterns based on geographical location of clients. Neither tool directly uses this notion of the geographical location in their alerting decisions, but rather it is mostly based on the tools making use of different known violator databases. We believe that the use of these different known violator databases is a major contributing factor to the overall diversity between these tools in the context of detecting malicious web scraping.

In contrast, the use of design diversity techniques in the context of using machine learning for malware classification shows much more promising results, as we highlight in our second section of work. We analysed a dataset of 37 different RNN machine learning models, and the predictions they generated regarding a sample dataset of over 4000 benign and malicious software samples. In our analysis, we discovered significant improvements in the performance of these machine learning models when used in design diversity combinations, under 1-out-of-N, N-out-of-N and simple majority adjudication schemes, with increases in sensitivity of up to 28 percentage points and specificity up to 14 points. Compared to our web scraping work, this dataset allows us a much bigger confidence in our results of diversity, due to the higher number of different models at our disposal. All 37 different RNN models performed noticeably different from one another, and were constructed using unique values of hyperparameters, such that each model was significantly diverse from every other model in their configuration.

This diversity in hyperparameters translated into diversity of alerting behaviour, and this was particularly noticeable with the different values of sequence length, which determined how long a model had to observe a sample's activity before it was asked to make a prediction. In our study we highlight clear evidence that different values of this sequence length parameter result in a high degree of diversity between the models' alerting patterns. We demonstrate this by highlighting how combinations where more extreme differences in sequence length value between the individual models exist result in higher sensitivity or specificity in overall systems. One of the major reasons we speculate for this is based on the mode of operation between various malware samples. Some malware samples generate incriminating activity early on in their execution cycle and the longer they continue their activity the less "damage" they generate, meaning that a model which generates its prediction earlier on will be more accurate at detecting it rather than a model that waits for longer and thus ends up considering more benign-looking activity as well. On the other hand, different malware samples could start their execution cycle less aggressively, perhaps setting up their attack before finally appearing malicious, in which case a model which takes very little time to generate a prediction might only have access to activity which looks benign. As opposed to our results with the two malicious web scraping detectors, the use of machine learning for malware classification appears to be an environment where design diversity in the models has a significant impact in the performance of security systems.

Finally, we detailed our approach on the application of optimal adjudication techniques in both malicious web scraping and malware classification contexts we analysed prior. Optimal adjudication extends beyond the use of conventional adjudication schemes (such as 1ooN or NooN). Inherently, optimal adjudication performs better than other adjudication schemes, by limiting the overall cost of generating false positive and false negative errors. This is because optimal adjudication makes use of the idea of syndromes, which partition a sample dataset into disjointed subsets, and makes predictions for each of these subsets based on historical data observed for them. This leads to lower costs than what other conventional adjudication schemes can achieve. Secondly, optimal adjudication brings with it a major benefit, which is the ability for a system administrator to be able to adjust the cost associated with the different error types (false positives and false negatives). In this manner, a system can be configured with the desired outcome of minimising overall error costs for the system, rather than relying on maximising sensitivity of specificity metrics, which may not be as important for an organisation looking to lower their security costs.

We have presented our methodology for the creation of optimal adjudication systems for both web scraping and machine learning model contexts. We have demonstrated the increased performance of optimal adjudication techniques compared to the more conventional adjudication schemes we presented in the individual studies, in particular with the RNN machine learning dataset. We do this with a particular focus on the ability of optimal adjudicators to base their predictions on adapting error costs for false positives and false negatives, which we believe is the defining feature which sets optimal adjudication apart from other schemes. When comparing optimal adjudicators built out of the RNN models with simple majority schemes, optimal adjudicators achieved an increased accuracy of 2 percentage points in combinations of 3 models, and 7 points in combinations of 5 models (when valuing false positives and false negatives the same). This demonstrates the ability of optimal adjudicators to improve in accuracy when more systems are added per combination, something that was not present in simple majority schemes.

When applying varying error costs, we demonstrate the adaptability of optimal adjudicators to minimise their overall cost, when compared with conventional adjudication schemes which show a linear increase in costs as error types are valued differently. Further, we expand these analysis to optimal adjudicators which make use of non-binary classifications, and the effects and downsides that applying different amounts of threshold points to these has on the resulting optimal adjudicator.

Finally, we have highlighted the different sources of uncertainty that are inherent with the creation of optimal adjudicators, in particular the uncertainty in training datasets and the uncertainty in the correct determination of error costs for different failure types.

7.1 Threats to validity and further work

Our work looking at the design diversity with web scraping detectors has several limitations that should be pointed out. Both tools, Arcane and CommTool, are treated as black boxes in our research. While we describe the diversity observed between them, this is mostly focused on the diversity of outcomes and their alerting patterns, and it is difficult to relate this back to the way the tools function. While we are unable to make general recommendations for the context of detection malicious web scraping activity, our results nonetheless have proved useful to our industrial partner Amadeus, who has made use of them to improve their detection infrastructure. We are unfortunately unable to make available the datasets which we analysed, due to our work relationship with Amadeus. The dataset provider is rightly concerned about disclosing information relating to their internal networks, and leaking information on how to bypass their web scraping detectors.

Further work in this area ought to be realised to overcome these limitations, by utilising a larger number of web scraping detectors. As was made clear with our work with RNN models, having a larger pool of potential systems is a very important aspect when it comes to diversity studies, and the potential results of a larger study would alleviate some of the major issues of our own work when it comes to the meaningfulness of the results. This would be especially true if commercially available or open-sourced software was included, as in these cases, one could make more generalised recommendations which would be more useful for practitioners beyond Amadeus.

Additionally, as we pointed out in our optimal adjudication section, the application of optimal adjudication techniques to the study of web scraping detectors is limited. The ability to use only two systems when creating an optimal adjudicator does not give us much flexibility in the construction of such a system, leaving us with too few possible syndromes to be able to significantly improve over more conventional adjudication schemes.

With regards to our work with RNN machine learning models for the classification of software samples, there are important aspects that future work in this area ought to address. One of these is the use of different types of machine learning models, as opposed to our work which has focused solely on the use of RNN architectures. While the models we analysed made use of a multitude of hyperparameters to diversify their operation, further diversification could easily be achieved by using different classifier models, with essentially no changes to the methodology and benchmarking of the various models. Further, additional activity metrics used by the models to analyse a software sample could be employed, in particularly the use of API calls, which is featured in separate bodies of work relating to the creation of machine learning models, but is notably absent from the dataset we have used for our analysis.

A secondary aspect which warrants additional research in this field concerns the analysis of detection rates based on malware types. While we have attempted an initial analysis based on malware types, the dataset we have worked with is limited due to a relatively low number of malware samples, and an uneven distribution across malware types. Work ought to be realised that takes into the account the performance of the various RNN models based on the types of malware samples they generate predictions for because, as we have pointed out multiple times throughout our discussions, the effects of design diversity are likely to vary between different classes of attacks. This could be expanded further by looking at different types of benign samples and how these affect the correct benign classification between the various models, as is the case with malware samples.

7.2 Review of research objectives

We believe that the research objectives we set out in chapter 1 were adequately met by the work we have presented in this thesis. We have (1) assessed the effects of design diversity on the reliability and security of systems, in two separate security contexts. In both we have presented the general improvements and drawbacks in performance of the various tools involved in the datasets analysed, and have delved deeper into their diversity aspects, highlighting the characteristics that led the tools to have diverse alerting patterns. We further expanded on these results by (2) developing a comprehensive study on the use of optimal adjudication, and its flexibility and improvements over other conventional adjudication schemes. For both of these points, we believe we have provided novel insight for the security community, as these works were performed in contexts which, to the best of our knowledge, has not been extensively studied in the past. Finally, we believe we have contributed to the professional and research communities by (3) developing methodology for the analysis of the effects of design diversity, supplying our industrial partner Amadeus with analysis visualisation dashboards, and detailing our procedures with the creation and development of optimal adjudication frameworks.

7.3 Final remarks

The use of design diversity for security is essential in the current age. While the continuous reliability improvement for individual tools is desired and should be maintained, it would be naive to set aside a fault tolerance technique which can often present easy solutions that have significant impacts on their performance. There has been a consistent increase in the number of off-the-shelf security software over the past few decades, and with no indication of this tendency to stop any time soon, the market is perfect for even smaller organisations to improve their security without relying on overly expensive solutions.

As we observe the effects that the COVID-19 pandemic has had on the security landscape, pushing almost every business and organisation online and into the cyber space, the security community should be incentivising the use of design diversity as a lower cost mitigation to the increased attack incentives that have also been created.

On the research side, more effort should be placed into the effects of design diversity, in particular, on developing methodologies and frameworks for assessing system diversity and the aspects that lead to tool diversity. We consider the work we have presented to be part of this crucial effort, and hope that the analysis and methodologies we presented herein can serve as a good foundation for future research endeavours.

8 References

[1] B. Littlewood, P. T. Popov, and L. Strigini, "Modeling software design diversity," *ACM Computer Surveys*, vol. 33, no. 2, pp. 177 – 208, 2001.

[2] DiSIEM project, "DiSIEM D3.2 - Probabilistic Modelling of Diversity for Security and Security Trends," 2018.

[3] N. Antunes and M. Vieira, "On the metrics for benchmarking vulnerability detection tools," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 505–516, 2015.

[4] M. R. Lyu *et al.*, *Handbook of software reliability engineering*, vol. 222. IEEE computer society press CA, 1996.

[5] B. Littlewood and L. Strigini, "Redundancy and diversity in security," in *9th European Symposium on Research in Computer Security*, vol. 3193, pp. 423 – 438, 2004.

[6] L. Strigini, "Fault tolerance against design faults," in *Dependable Computing Systems: Paradigms, Performance Issues, and Applications*, pp. 213 – 241, John Wiley & Sons, 2005.

[7] P. Traverse, "Airbus and atr system architecture and specification," in *Software diversity in computerized control systems*, pp. 95–104, Springer, 1988.

[8] H. Anderson and G. Hagelin, "Computer controlled interlocking system," *Ericsson Review*, vol. 2, pp. 74–80, 1981.

[9] K. Kanoun, "Cost of software design diversity an empirical evaluation," in *Proceedings 10th International Symposium on Software Reliability Engineering*, pp. 242–247, Nov 1999.

[10] Symantec, "2019 Internet Security Threat Report - ISTR Volume 24," 2019.

[11] D. project, "DiSIEM project - Diversity Enhancements for Security Information and Event Management."

[12] M. Rhode, P. Burnap, and K. Jones, "Early-stage malware prediction using recurrent neural networks," *Computers & Security*, vol. 77, pp. 578 – 594, 2018.

[13] P. Marques, Z. Dabbabi, M.-M. Mironescu, O. Thonnard, A. Bessani, F. Buontempo, and I. Gashi, "Detecting malicious web scraping activity: a study with diverse detectors," in *The 23rd IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2018)*, October 2018.

[14] P. Marques, Z. Dabbabi, M.-M. Mironesc, O. Thonnard, A. Bessan, F. Buontempo, and I. Gashi, "Using diverse detectors for detecting malicious web scraping activity," *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 67–68, 2018.

[15] P. Marques, M. Rhode, and I. Gashi, "Waste not: Using diverse neural networks from hyperparameter search for improved malware detection," *Computers & Security*, vol. 108, p. 102339, 2021.

[16] B. Kitchenham, "Procedures for performing systematic reviews," *Keele, UK, Keele Univ.*, vol. 33, 08 2004.

[17] A. S. Nascimento, C. M. F. Rubira, R. Burrows, and F. Castor, "A systematic review of design diversity-based solutions for fault-tolerant soas," in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, EASE '13, (New York, NY, USA), pp. 107–118, ACM, 2013.

[18] D. Acarali, M. Rajarajan, N. Komninos, and I. Herwono, "Survey of approaches and features for the identification of http-based botnet traffic," *Journal of Network and Computer Applications*, October 2016.

[19] P. Vinod, R. Jaipur, V. Laxmi, and M. Gaur, "Survey on malware detection methods," in *Proceedings of the 3rd Hackers' Workshop on computer and internet security (IITKHACK'09)*, pp. 74–79, 2009.

[20] I. Saeed, A. Selamat, and A. Abuagoub, "A survey on malware and malware detection systems," *International Journal of Computer Applications*, vol. 67, pp. 25–31, 04 2013.

[21] David Powell and Robert Stroud, "Conceptual model and architecture of MAFTIA," 2003.

[22] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11–33, Jan 2004.

[23] B. Littlewood, S. Brocklehurst, N. E. Fenton, P. Mellor, S. Page, D. Wright, J. Dobson, J. McDermid, and D. Gollmann, "Towards operational measures of computer security," *Journal of Computer Security*, vol. 2, pp. 211–230, 1993.

[24] A. Avizienis and L. Chen, "On the implementation of n-version programming for software fault tolerance during program execution," 01 1997.

[25] A. Condor and G. Hinton, "Fault tolerant and fail-safe design of candu computerised shutdown systems," in *IAEA Specialist Meeting on Microprocessors important to the Safety of Nuclear Power Plants*, 1988.

[26] G. Dahll and J. Lahti, "An investigation of methods for production and verification of highly reliable software," in *Safety of Computer Control Systems*, pp. 89–94, Elsevier, 1980.

[27] J. P. Kelly and A. Avizienis, "A specification-oriented multi-version software experiment," in *Digest of Papers FTCS-13: Thirteenth International Conference on Fault Tolerant Computing*, pp. 120–125, 1983.

[28] A. Avizienis, M. R. Lyu, and W. Schutz, "In search of effective diversity: a six-language study of fault-tolerant flight control software," in *The Eighteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pp. 15–22, June 1988.

[29] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Transactions on software engineering*, no. 1, pp. 96–109, 1986.

[30] D. E. Eckhardt, A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk, and J. P. J. Kelly, "An experimental evaluation of software redundancy as a strategy for improving reliability," *IEEE Transactions on Software Engineering*, vol. 17, pp. 692–702, July 1991.

[31] D. E. Eckhardt and L. D. Lee, "A theoretical basis for the analysis of multiversion software subject to coincident errors," *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 1511–1517, Dec 1985.

[32] J. C. Knight and N. G. Leveson, "A reply to the criticisms of the knight & leveson experiment," *SIGSOFT Softw. Eng. Notes*, vol. 15, pp. 24–35, Jan. 1990.

[33] A. Avizienis, "The methodology of n-version programming," in *Software fault tolerance* (M. R. Lyu, ed.), ch. 2, John Wiley & Sons Ltd., 1995.

[34] L. L. Pullum, "A new adjudicator for fault tolerant software applications correctly resulting in multiple solutions," in *AIAA/IEEE Digital Avionics Systems Conference*, pp. 147–152, Oct 1993.

[35] S. S. Brilliant, J. C. Knight, and N. G. Leveson, "The consistent comparison problem in n-version software," *IEEE Transactions on Software Engineering*, vol. 15, pp. 1481–1485, Nov 1989.

[36] Cisco systems, "Snort," 2021. https://www.snort.org/.

[37] Open Information Security Foundation, "Suricata," 2021. https://suricata-ids.org/.

[38] Vern Paxson, "Zeek," 2021. https://www.zeek.org/.

[39] H. Asad and I. Gashi, "Dynamical analysis of diversity in rule-based open source network intrusion detection systems," *Empirical Software Engineering*, September 2021.

[40] A. Algaith, I. A. Elia, I. Gashi, and M. R. Vieira, "Diversity with intrusion detection systems: An empirical study," in *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, IEEE, December 2017.

[41] I. Gashi, V. Stankovic, C. Leita, and O. Thonnard, "An experimental study of diversity with off-the-shelf antivirus engines," in *Eighth IEEE International Symposium on Network Computing and Applications*, pp. 4 – 11, 2009.

[42] C. Leita and M. Dacier, "Sgnet: A worldwide deployable framework to support the analysis of malware threat models," in *2008 Seventh European Dependable Computing Conference*, pp. 99–109, May 2008.

[43] V. Stankovic, R. E. Bloomfield, P. G. Bishop, and I. Gashi, "Diversity for security: a study with off-the-shelf antivirus engines," in *21st International Symposium on Software Reliability Engineering (ISSRE 2011)*, IEEE Computer Society Press, 2011.

[44] I. Gashi, P. T. Popov, and L. Strigini, "Fault diversity among off-the-shelf sql database servers," in *International Conference on Dependable Systems and Networks*, pp. 389 – 398, 2004.

[45] M. Garcia, A. N. Bessani, I. Gashi, N. Neves, and R. R. Obelheiro, "Analysis of operating system diversity for intrusion tolerance," *Software: Practice and Experience*, 2013.

[46] D. Opitz and R. Maclin, "Popular ensemble methods: An empirical study," vol. 11, 12 1999.

[47] F. Di Giandomenico and L. Strigini, "Adjudicators for diverse-redundant components," in *Proceedings Ninth Symposium on Reliable Distributed Systems*, pp. 114–123, Oct 1990.

[48] D. M. Blough and G. F. Sullivan, "A comparison of voting strategies for fault-tolerant distributed systems," in *Proceedings Ninth Symposium on Reliable Distributed Systems*, pp. 136–145, Oct 1990.

[49] J. Yerushalmy, "Statistical problems in assessing methods of medical diagnosis, with special reference to x-ray techniques," *Public Health Reports (1896-1970)*, vol. 62, no. 40, pp. 1432–1449, 1947.

[50] B. Matthews, "Comparison of the predicted and observed secondary structure of t4 phage lysozyme," *Biochimica et Biophysica Acta (BBA) - Protein Structure*, vol. 405, no. 2, pp. 442 – 451, 1975.

[51] D. Powers, "Evaluation: From precision, recall and f-factor to roc, informedness, markedness & correlation," *Mach. Learn. Technol.*, vol. 2, 01 2008.

[52] G. Benford, *The Scarred Man.* 1970.

[53] F. Cohen, "Computer viruses: Theory and experiments," *Computers & Security*, vol. 6, no. 1, pp. 22–35, 1987.

[54] J. Aycock, *Computer viruses and malware*, vol. 22. Springer Science & Business Media, 2006.

[55] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 377–396, 2016.

[56] G. Canfora, A. Di Sorbo, F. Mercaldo, and C. A. Visaggio, "Obfuscation techniques against signature-based detection: A case study," in *2015 Mobile Systems Technologies Workshop (MST)*, pp. 21–26, 2015.

[57] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *Computers & Security*, vol. 28, no. 1, pp. 18–28, 2009.

[58] I. Firdausi, C. lim, A. Erwin, and A. S. Nugroho, "Analysis of machine learning techniques used in behavior-based malware detection," in *2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies*, pp. 201–203, 2010.

[59] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of machine learning.* MIT press, 2018.

[60] H. El Merabet and A. Hajraoui, "A survey of malware detection techniques based on machine learning," *Int. J. Adv. Comput. Sci. Appl*, vol. 10, no. 1, pp. 366–373, 2019.

[61] M. Masud, L. Khan, and B. Thuraisingham, "A hybrid model to detect malicious executables," pp. 1443–1448, 06 2007.

[62] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq, "Pe-miner: Mining structural information to detect malicious executables in realtime," in *Recent Advances in Intrusion Detection* (E. Kirda, S. Jha, and D. Balzarotti, eds.), (Berlin, Heidelberg), pp. 121–141, Springer Berlin Heidelberg, 2009.

[63] M. Siddiqui, M. Wang, and J. Lee, "Detecting trojans using data mining techniques," pp. 400–411, 04 2008.

[64] J. Bai, J. Wang, and G. Zou, "A malware detection scheme based on mining format information," *TheScientificWorldJournal*, vol. 2014, p. 260905, 06 2014.

[65] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, "Malware detection by eating a whole exe," 2017.

[66] M. Abdelsalam, R. Krishnan, Y. Huang, and R. Sandhu, "Malware detection in cloud infrastructures using convolutional neural networks," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 162–169, 2018.

[67] C. T. Dan Lo, O. Pablo, and C. M. Carlos, "Towards an effective and efficient malware detection system," in *2016 IEEE International Conference on Big Data (Big Data)*, pp. 3648–3655, 2016.

[68] SonicWall, "2020 SonicWall Cyber Threat Report - Mid-year update," 2020.

[69] A. Balci, D. Ungureanu, and J. Vondruška, "Malware reverse engineering handbook," 2020.

[70] T. L. group, "To kill a centrifuge," 2013.

[71] Interpol, "COVID-19 cyberthreats," 2021. https://www.interpol.int/en/Crimes/Cybercrime/COVID-19-cyberthreats.

[72] Deloitte, "Impact of COVID-19 on Cybersecurity," 2021. https://www2.deloitte.com/ch/en/pages/risk/articles/impact-covid-cybersecurity.html.

[73] OWASP foundation, "OWASP foundation," 2021. https://owasp.org/.

[74] OWASP foundation, "A1:2017-Injection," 2021. https://owasp.org/www-project-top-ten/2017/A1_2017-Injection.

[75] J. Clarke-Salt, *SQL injection attacks and defense*. Elsevier, 2009.

[76] D. Stuttard and M. Pinto, *The web application hacker's handbook: Finding and exploiting security flaws*. John Wiley & Sons, 2011.

[77] N. Antunes and M. Vieira, "Penetration testing for web services," *Computer*, vol. 47, no. 2, pp. 30–36, 2014.

[78] N. Antunes and M. Vieira, "Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services," in *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, pp. 301–306, 2009.

[79] W. G. J. Halfond and A. Orso, "Preventing sql injection attacks using amnesia," in *ICSE '06*, 2006.

[80] OWASP foundation, "A7:2017-Cross-Site Scripting (XSS)," 2021. https://owasp.org/www-project-top-ten/2017/A7_2017-Cross-Site_Scripting_(XSS).html.

[81] OWASP, "Stored cross-site scripting," 2021. https://owasp.org/www-community/attacks/xss/#stored-xss-attacks.

[82] V. K. Malviya, S. Saurav, and A. Gupta, "On security issues in web applications through cross site scripting (xss)," in *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, vol. 1, pp. 583–588, 2013.

[83] OWASP foundation, "Vulnerability Scanning Tools," 2021. https://owasp.org/www-community/Vulnerability_Scanning_Tools.

[84] Michael Schrenk, *Webbots, Spiders and Screen scrapers: a guide to developing internet agents with PHP/CURL*. No starch press, 2007.

[85] M. Bhatia and D. Gupta, "Discussion on web crawlers of search engine," *COIT-2008*, 2008.

[86] J. Hillen, "Web scraping for food price research," *British Food Journal*, 2019.

[87] C. Slamet, R. Andrian, D. S. Maylawati, Suhendar, W. Darmalaksana, and M. A. Ramdhani, "Web scraping and naïve bayes classification for job search engine," vol. 288, p. 012038, jan 2018.

[88] DigitalMR, “Digitalmr,” 2021. https://www.digital-mr.com/.

[89] Y. Sun, I. G. Councill, and C. L. Giles, “The ethicality of web crawlers,” in *2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, vol. 1, pp. 668–675, 2010.

[90] Igal Zeiffman, “Bot traffic report 2016,” 2021. https://www.imperva.com/blog/bot-traffic-report-2016/?redirect=Incapsula.

[91] Y. Alnoamany, M. Weigle, and M. Nelson, “Access patterns for robots and humans in web archives,” *Proceedings of the ACM/IEEE Joint Conference on Digital Libraries*, 09 2013.

[92] I. I. Savchenko and O. Y. Gatsenko, “Analytical review of methods of providing internet anonymity,” *Automatic Control and Computer Sciences*, vol. 49, pp. 696–700, Dec 2015.

[93] N. Geens, J. Huysmans, and J. Vanthienen, “Evaluation of web robot discovery techniques: A benchmarking study,” in *Advances in Data Mining. Applications in Medicine, Web Mining, Marketing, Image and Signal Mining* (P. Perner, ed.), (Berlin, Heidelberg), pp. 121–130, Springer Berlin Heidelberg, 2006.

[94] Weigang Guo, Shiguang Ju, and Yi Gu, “Web robot detection techniques based on statistics of their requested url resources,” in *Proceedings of the Ninth International Conference on Computer Supported Cooperative Work in Design, 2005.*, vol. 1, pp. 302–306 Vol. 1, 2005.

[95] P.-N. Tan and V. Kumar, *Discovery of Web Robot Sessions Based on Their Navigational Patterns*, pp. 193–222. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[96] A. M. TURING, “I.—COMPUTING MACHINERY AND INTELLIGENCE,” *Mind*, vol. LIX, pp. 433–460, 10 1950.

[97] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford, “Captcha: Using hard ai problems for security,” in *Advances in Cryptology — EUROCRYPT 2003* (E. Biham, ed.), (Berlin, Heidelberg), pp. 294–311, Springer Berlin Heidelberg, 2003.

[98] K. Park, V. S. Pai, K.-W. Lee, and S. Calo, “Securing web service by automatic robot detection,” in *Proceedings of the Annual Conference on USENIX ’06 Annual Technical Conference*, ATEC ’06, (USA), p. 23, USENIX Association, 2006.

[99] VirusTotal, “VirusTotal,” 2021. https://www.virustotal.com.

[100] SenseCy, “SenseCy,” 2021. https://www.sensecy.com/.

[101] SurfWatch Labs, “SurfWatch,” 2021. https://www.surfwatchlabs.com/.

[102] Cisco, “Cisco Security Advisory,” 2021. https://tools.cisco.com/security/center/publicationListi

[103] Threatpost, “Threatpost,” 2021. https://threatpost.com/.

[104] National Institute of Standards and Technology, “National Vulnerability Database,” 2021. https://nvd.nist.gov/.

[105] C. Sabottke, O. Suciu, and T. Dumitras, “Vulnerability disclosure in the age of social media: Exploiting twitter for predicting real-world exploits,” in *24th USENIX Security Symposium (USENIX Security 15)*, (Washington, D.C.), pp. 1041–1056, USENIX Association, Aug. 2015.

[106] S. Mittal, P. K. Das, V. Mulwad, A. Joshi, and T. Finin, “Cybertwitter: Using twitter to generate alerts for cybersecurity threats and vulnerabilities,” in *Proceedings of the 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ASONAM ’16, (Piscataway, NJ, USA), pp. 860–867, IEEE Press, 2016.

[107] R. Campiolo, L. A. F. Santos, D. M. Batista, and M. A. Gerosa, “Evaluating the utilization of twitter messages as a source of security alerts,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC ’13, (New York, NY, USA), pp. 942–943, ACM, 2013.

[108] DiSIEM project, “DiSIEM D4.2 - OSINT data fusion and analysis architecture,” 2018.

[109] MISP project, “MISP - Malware Information Sharing Platform,” 2021. https://www.misp-project.org/.

[110] I. You and K. Yim, “Malware obfuscation techniques: A brief survey,” in *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, pp. 297–300, 2010.

[111] L. Nataraj, V. Yegneswaran, P. Porras, and J. Zhang, “A comparative assessment of malware classification using binary texture analysis and dynamic analysis,” in *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, AISec ’11, (New York, NY, USA), p. 21–30, Association for Computing Machinery, 2011.

[112] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, “Adversarial examples for malware detection,” in *Computer Security – ESORICS 2017* (S. N. Foley, D. Gollmann, and E. Snekkenes, eds.), (Cham), pp. 62–79, Springer International Publishing, 2017.

[113] A. Damodaran, F. Di Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, “A comparison of static, dynamic, and hybrid analysis for malware detection,” *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 1, pp. 1–12, 2017.

[114] M. Rhode, L. Tuson, P. Burnap, and K. Jones, “Lab to soc: Robust features for dynamic malware detection,” in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks–Industry Track*, pp. 13–16, IEEE, 2019.

[115] J. Bergstra and Y. Bengio, “Random search for hyper-parameter optimization,” *J. Mach. Learn. Res.*, vol. 13, pp. 281–305, Feb. 2012.

[116] N. S. Keskar and R. Socher, “Improving generalization performance by switching from adam to sgd,” 2017.

[117] M. Nashaat, A. Ghosh, J. Miller, and S. Quader, “Asterisk: Generating large training datasets with automatic active supervision,” *ACM/IMS Trans. Data Sci.*, vol. 1, May 2020.