TCPMISSING: AN INTELLIGENT ANALYTICAL COMPONENT FOR THE DETERMINATION OF MISSING PACKETS

by

REBEKAH BLACK

(Under the Direction of Walter D. Potter)

ABSTRACT

TCPMissing is a program that evaluates a TCP trace file to determine the number of missing packets caused by the application because the system it is running on is not fast enough to capture, process, or store all the packets. Although packet loss information is generally computed and reported to the capture program, this information does not get stored and distributed with the file. An important tool in traffic analysis would be an application that could take an input file and determine the average loss rate caused by the capture program. In addition to implementing a sequential algorithm to solve this problem, genetic programming is used to develop a mathematical model to intelligently solve the problem of missing packets. The discovered equation is implemented and tested in order to compare the performance of both methods in computing the number of missing packets.

INDEX WORDS: Artificial Intelligence, Computer Science, Intrusion Detection, Network Security, Genetic Programming, Transmission Control Protocol, Evolutionary Computing in Java, Java Packet Capture

TCPMISSING: AN INTELLIGENT ANALYTICAL COMPONENT FOR THE DETERMINATION OF MISSING PACKETS

by

REBEKAH BLACK

BS, The University of Georgia, 2006

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2006

TCPMISSING: AN INTELLIGENT ANALYTICAL COMPONENT FOR THE DETERMINATION OF MISSING PACKETS

by

REBEKAH BLACK

DEDICATION

"'It could be a torture chamber or a dungeon or a hideous pit or anything!'

'It's just a student's bedroom, sergeant.'

'You see?'" (Terry Pratchett, "Men At Arms")

I would like to dedicate this thesis to the main philosophers who shaped my thinking during my time as a graduate student: Kahlil Gibran and Terry Pratchett. The college experience is a time of stretching, and during this period I found the thoughts of Gibran and Pratchett particularly refreshing and endearing, and even humorous. To a large extent, the adoption of their views affects the way I approach life, even the way I approached researching and writing my thesis. The reader of my thesis may recognize the influence of Gibran or Pratchett. In the words of Gibran,

"And I say that life is indeed darkness save when there is urge,

And all urge is blind save when there is knowledge,

And all knowledge is vain save when there is work,

And all work is empty save when there is love;

And when you work with love you bind yourself to yourself, and to one another, and to God." ("The Prophet")

ACKNOWLEDGEMENTS

This thesis was hard. I would like to thank several people for their help to me.

First I would like to thank Don Potter for being my advisor. And for inspiring an interest in me for the field of AI in my first years at UGA. And for continuing to mentor and advise me along the educational path.

I would like to thank Dan Everett for inspiring me in the field of network security during my first years at UGA. And for all those age-appropriate beverages ☺ along the way!

I would like to thank Kang Li for giving me the platform and knowledge and inspiration to do much of this project. It was during his network security class that I first began this project, and without his intellectual guidance, I would never have had the tools to finish it.

I would like to thank Kris Kochut for the departmental assistantships. I have really cherished the opportunity to teach 1301 students during my time as a grad student. And the financial assistance was much appreciated!

I would like to thank Hamid Arabnia for his thoughtful TA assignments. I always ended up with just the right responsibilities each semester. In addition to those duties, Dr. Arabnia went the extra mile with his open door policy. There was never a moment when he was too busy. He made time to give students his full attention, and his advice always demonstrates his innate understanding of the bigger picture.

Brian Smith was an invaluable AI technique resource. He knows his stuff in and out and is also an excellent and patient teacher. He could intuit exactly what I did not understand yet and always knew what direction to point me in.

Mayukh Dass proved to be a huge mentor along the way, always helpful and supportive. I am always impressed by the thoughtful advice he has to offer on all my queries. There is nothing I am doing he has not already done, and done well! Thanks for being such an inspiration!

I would like to extend my deep appreciation to Dorothy Denning, Rebecca Bace, and Raven Alder for giving me of their time and allowing me to conduct extensive telephone interviews with them. They were generous in sharing their wisdom and experiences with me, all of which contributed greatly to this endeavor. I hope our paths cross again in the future.

I would like to thank my grandfather for his financial contributions to “Project Rebekah Black” over the years. I could never have made it this far without you!

My mom and dad have always been supportive of my goals and dreams. I am glad that they have been able to live close by in this period of my life so that they could share in much of my experiences. I appreciate how much you put up with in this tumultuous, changing period of my life.

TABLE OF CONTENTS

LIST OF FIGURES

GLOSSARY OF TERMS AND ACRONYMS

ACK	Acknowledgment
AI	Artificial Intelligence
ANN	Artificial Neural Network
API	Application Program Interface
ARP	Address Resolution Protocol
BGP	Border Gateway Protocol
CS	Computer Science
DIDS	Distributed IDS
ENIAC	Electronic Numerical Integrator Analyzer and Computer
EOT	End of Transmission
ES	Expert System
FIN	Finish
GA	Genetic Algorithm
GP	Genetic Programming
HIDS	Host-based IDS
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDS	Intrusion Detection System
IP	Internet Protocol
IPSEC	Internet Protocol Security
IT	Information Technology
JPCAP	Java Packet CAPture
NIDS	Network-based IDS
OOPL	Object-Oriented Programming Language
OS	Operating System
SAR	Segmentation and Reassembly
SciTech	US Gov's Science and Technology
SEQ	Sequence Number
SIDS	Storage-based IDS
SPAWAR	SPAce and WARfare
SRI	Stanford Research Institute
SYN	Synchronize
TCP	Transmission Control Protocol

FORWARD

Intrusion detection systems have evolved into complex and intricate creatures. They have been given this opportunity because there is an ever increasing market for security solutions from an ever growing Internet threat. Even though many different efficient intrusion detection systems currently exist, it is easy for the setup and configuration to be forgotten or ignored by system administrators. In addition, new attacks are constantly being engineered that can bypass system security. Partially because of these reasons, and the sometimes slow response of humans as compared to computers, artificial intelligence techniques have become popular in the intrusion detection domain.

The basic premise behind real-time intrusion detection is to analyze packets for malicious patterns as they come off the wire. In addition to performing this check as the first line of defense, it is also common to store network traffic in files for possible later analysis in case post-attack forensics becomes necessary.

Files containing network traffic information essentially contain all the packets passing through the network in a certain time frame. These files can be stored either as binary or as text files, and they are usually referred to as “dump” or “trace” files.

As Internet speeds increase, it has become possible for the application performing the capture of the packets to have trouble keeping up. When that happens, certain packets may go missing from the dump file because the system was not fast enough to capture, process, or store all the packets. This is opposed to packets that may be dropped at the network level. Packets dropped at the network level may be lost due to network conditions such as congestion. Those packets never make it to their destination and require retransmission whereas “missing” packets do not because they are only missing from the trace file, not the actual network traffic stream.

Although packet loss information is generally computed and reported to the capture program, this information does not get stored and distributed with the file. An important tool in traffic analysis would be an application that could take an input file and determine the number of missing packets caused by the capture program. Knowing this information is important because it would affect the manner in which post-attack forensics is conducted. Knowing that the full amount of data is available at their fingertips would allow investigators to proceed as usual. However, being aware that some packets may be missing from the trace file will alert investigators to possible loss of data and help prevent them from reaching erroneous conclusions based on an incomplete dataset.

TCPMissing is an enterprising application that receives as input a trace file and calculates the number of missing packets. The TCPMissing algorithmic solution is elucidated in detail in chapter two. In addition to developing an algorithmic solution, genetic programming was used to engineer a mathematical equation as a solution to this problem. The mathematical equation resulting from genetic programming contains variables representing packet values from the input file and operators which manipulate those variables in some way. The result obtained from evaluating the equation signifies whether a packet has gone missing or not.

Genetic programming often arrives at surprising and accurate solutions to often very difficult problems. It helps to have a well-defined problem with known solutions to set as the fitness function for the evolutionary computations. In this case, with files of packets, it is easy to remove selected packets thus knowing exactly where and how many were removed. The genetic programming approach is described in chapter three.

Chapter one discusses the development of network security as a discipline and how previous research done in this field pointed the way towards TCPMissing. It specifically focuses in on the work of three key female figures in network security and shows how their insights capture the overarching mentality permeating the security field today. It was through exposure to these women’s experiences, contributions, and advice that the idea for a genetic programming solution to TCPMissing arose.

CHAPTER 1

INTRUSION DETECTION

Intrusion detection systems have been around the block several times. An intrusion detection system is essentially a program whose purpose is to detect either anomalous or malicious behavior and report it though the proper channels, usually to alert the system administrator so that appropriate counter-action can be taken. This is opposed to intrusion prevention measures which consist of now standard computer security practices such as smart password choosing. The first section of this chapter highlights in more detail the importance of developing this field of study.

A monumental paper written by James Anderson in 1980 touches off discussion concerning intrusion detection systems as we know them. Anderson's paper covered basic audit trails and how they were good indicators of any suspicious behavior going on in the system [Ande1980]. However, Anderson conceded that even the audit trail may be subverted by a malicious user. This was really the first paper of its type, even though it did not explicitly use the term "intrusion detection" and it did not detect intrusions in real time.

Stanford Research Institute (SRI) became interested in intrusion detection systems in 1984. SRI is a Californian-based independent non-profit research institute that conducts research for clients such as government or private agencies. The US government sponsored

SRI’s Dorothy Denning to conduct research on intrusion detection systems and develop a model for future implementations. The Navy contracted with SRI to produce the first IDS implementation IDES (Intrusion Detection Expert System). Denning was not even aware of James Anderson’s paper at the time of her work with intrusion detection.

In addition to Dorothy Denning, Rebecca Bace and Raven Alder have also contributed to the development of the IDS field. The second section of this chapter highlights their experience working in this field and the contributions and impact they have made. Not only has their work influenced the field of intrusion detection, but their insight and advice has, to a certain degree, determined the topic and writing of this thesis. Much of the information presented was gleaned from recent telephone interviews conducted first hand with the three women mentioned above.

The last section of this chapter wraps up this basic survey of intrusion detection by presenting some of the most recent work done in the IDS field. The specific examples presented were chosen because they also represent some of the most creative techniques found to counter increasingly cunningly crafted intrusion threats.

The Importance of Intrusion Detection

As the Internet increases in importance in the role it plays in our culture, for commerce, communication, and even entertainment, it also increases in its propensity to become a target, whether it be for a political activist making a statement or a teenager exploring his world.

Network attacks are common and prolific. It is increasingly difficult to write software to detect attacks as the creativity and determination of attackers increases. As information technology becomes ever more capable of representing an individual's identity online, the chance for abuse also increases. Identity theft has become a common and real threat.

Many aspects of information technology make everyday tasks much easier. Online banking saves time and money. Online shopping is extremely quick and convenient for the buyer who knows exactly what they want. Large amounts of information can now be catalogued and stored to be called back up near-instantaneously at the touch of a button. IRS records, school records, even medical records are stored on computers. Having information in this form increases the productivity and efficiency of businesses, and that has contributed to the spread and proliferation of computer technology. And yet, the chance for abuse also increases.

"For every action, there is an equal and opposite reaction," states Newton's second law. If hackers have reacted to the spread of information technology by attempting to subvert systems into performing actions designers originally did not intend, then intrusion detection is in turn a counter-reaction to that trend. Not only does intrusion prevention attempt to counter *known* intrusion techniques, but it also seeks to preemptively strengthen systems against *unknown* intrusions.

Intrusion detection recognizes that the toughest guard might not catch every single attack, and attempts to provide ways to realize an attack is taking place and notify the administrator who then in turn can take measures to address the situation. These measures may include stopping the current attack maybe by denying the attacker access to the attacked system or by completely removing the system from access to the Internet, at least until the weakness is identified.

Once the weakness is identified, the administrator attempts to strengthen the system, maybe by creating/changing passwords for the system, or fixing application/system vulnerabilities that allowed the original break-in to occur.

The administrator may even go so far as to attempt to discover the identity of the perpetrator. This might be useful for preventing future attacks by the same savvy attacker. In the short term, detaining or incarcerating an attacker might prevent the immediate spread of ideas or techniques used in the original attack. However, in the long term, these ideas are likely to be repeated whether by the same individual or developed independently by another. So this method of intrusion prevention is less favorable than that of taking technical steps to strengthen the cyber-security aspects of a system.

All these steps of dealing with a security breach of a computer system are dependent upon correctly identifying an intrusion in the first place. This is why intrusion detection has risen to such an important place in our society. The role intrusion detection has been given is “equal and opposite” to that of intrusion itself in that they are both necessary to push the bounds of computer science forward at such an exhilarating pace. No one would argue that they are not opposite, although the equality might be debated. The term “equal” is used here in the sense of balancing forces, that combined together, have the effect of driving forward humanity’s digital acumen. It is not to be confused with the sense of the term “equal” that refers to a value-based appropriation of this culture’s technical know-how. Obviously, the hacker’s set of skills often meet with the general disapproval of Western culture due to the often detrimental effects that may arise from the practice of his black art.

The scrutiny of roles assigned to various segments of the population is an interesting and sometimes controversial subject to study. The last sentence of the paragraph above contains reference to a third person possessive singular personal pronoun. It is also masculine. This was intentionally done to illustrate the much-remarked upon tendency for the hacker population to be comprised of males, if not necessarily young nor white any more [vanM2000].

An interesting correlation to this point is the fact that women do seem to crop up in the history of intrusion detection. Perhaps this is part of society's intricate set of checks and balances to help keep the wheels turning smoothly. This issue is addressed in more detail in the next section after presenting three current examples of women doing intrusion detection and doing it well.

Key Players in Intrusion Detection

The purpose of this section is to outline a few of the key players in the history of intrusion detection systems, written in such a way as to demonstrate the impact of women in intrusion detection and their influence upon current research. Specifically, the women that have been focused on and interviewed are the following: Dorothy Denning, Rebecca Bace, and Raven Alder.

Dorothy Denning first did work in real time intrusion detection system back in 1984, and the Navy did most of the funding. Teresa Lunt is also a big name in the intrusion detection field and eventually was hired on by Denning to help with her research. Lunt was on the team that did the first implementation of Denning's initial model. That was back when intrusion detection was first coming into being, and Denning and Lunt both played large roles in that early stage. Currently, there are various women involved in researching intrusion detection (Raven Alder) and even women CEO's heading up commercial intrusion detection companies (Rebecca Bace).

This section also takes a brief look at why women go into the field of intrusion detection. Traditionally the field of security has belonged to men, and even recently computer science still seems to be something of a male discipline, especially when looking at percentages of doctoral degrees earned by men verses women. In fact, at the rate women are earning computer science doctoral degrees, parity with men will not be reached until the academic year 2087-88, more than 80 years from now [Mosk2002]. This runs counter to the observation that many women have been involved in the development of intrusion detection, from its conception in the early eighties till today where even more women have joined the scene. This section explores the specific reasons of three women for entering the field of intrusion detection. It also generalizes on why there may be the propensity for women to be drawn to intrusion detection, as opposed to some other area of computer science.

Since there is greater cultural viscosity to hamper the entry of women onto the computer science scene, women who do stick within the discipline are often very good at what they do, or at least the most stubborn. In essence, social factors weed out all but the most determined women. That is what prompted these three case studies. It was hoped that women who had defeated the odds by penetrating a largely male discipline such as network security within computer science would also be aware of the factors that led to their success and the underpinning mentality of the security community, knowledge which could bring about greater insight into the bigger picture and focus research on critical areas.

Dorothy Denning

In 1984, Stanford Research Institute (SRI) became interested in Intrusion Detection Systems. SRI is a Californian-based independent non-profit research institute that conducts research for clients such as government or private agencies. The US government sponsored SRI's Dorothy Denning to conduct research on intrusion detection systems and develop a model for future implementations. The Navy contracted with SRI to produce the first IDS implementation IDES (Intrusion Detection Expert System). Denning was not yet aware of James Anderson's paper on audit analysis at the time.

When Denning came to SRI, some work was already going on in audit analysis, and she participated in that. After she was there for a little bit, Karl Levitt (Associate Director of the lab, now at UC Davis) became interested in pursuing intrusion detection systems, and she was assigned to that field. SPAWAR and H. O. Lubdis sponsored their work at the time. Lubdis and Levitt had the initial notion of doing intrusion detection using expert systems. While Denning was at SRI, Levitt handed the project to her and Peter Neumann, but Denning was the one with the vision of wanting to do it in real-time instead of using expert systems to analyze audit systems after-the-fact. They all put together the proposal and SPAWAR ended up providing funding for the design of a conceptual model. Dorothy Denning was a principle investigator on what became the model project for intrusion detection systems [Denn1987, model]. Anderson's work was mostly just audit material analysis. The intrusion detection system work Denning did focused on real-time identification, which contributed greatly to that field [Denn2006].

In other circles, people were working on network intrusion detection systems. After Denning and her team got the project going, they were able to look beyond SRI to see what other people were doing in similar veins. At SciTech, Teresa Lunt was doing work in 1985 concerning network intrusion detection systems. Denning hired and then rehired Lunt to come over to SRI and work on her IDS projects [Denn2006].

Denning got involved in security for two reasons: it was interesting to her and had ideas that generated interest among those with funding power. She has remained in the area of network security because it has proven to be a rich and lucrative field to someone with her talent and ideas. Her goals have always been in the broader sense of upholding high moral standards of honesty and excellence, which are both good attributes to hold when working in the area of computer security [Denn2006].

Rebecca Bace

Even back in the late eighties, Denning's work was already on the radar, but Bace did not meet Denning till around the 1990 timeframe. It was still not a huge research community at that point, and so researchers could still enjoy things like the national conference (a get together of the three hundred people in the computer security research community, and then another three hundred government people to cover those at the conference [Bace2006]).

It was the most serendipitous of rationale that got Bace into computer security. (Bace's partner at Infidel, Terri Gilbert, says that serendipity is what happens when one consciously makes a piece of oneself available-things do converge. “It's amazing how things converge over time [Thie2002].”) Bace originally thought she would get a degree in civil engineering at University of Alabama, but she stopped short because she hated thermodynamics. She kept taking little side-trips on her path to her degree because she truly did not want to continue down that path. She knew she wanted to do something engineering-wise but had not yet discovered her niche. “She was born to be an engineer, but in denial about the whole thing,” says her husband Paul. She finally graduated with a degree in Computer Science and even went back to get a master’s in CS with an emphasis in system engineering. It was very practicum oriented at the time, an overarching engineering degree that focused on industrial techniques, like the thermodynamics class in question. Later as a graduate student she went back and re-did thermodynamics coursework in physics in desperation, just to prove to herself that she could do it. The people teaching thermodynamics at the time were steam-table guys, working for US Steel there in Birmingham, and it seemed hopelessly brain-dead to Bace. They were definitely not pros on the teaching end, they were practicing engineers, very much industrial engineers, and it was a different level of practice from academia. She was the only woman in the program and it was a bizarre situation. It was also a ticket out. Bace left the civil engineering program [Bace2006].

She finished her undergrad through a remote situation through New York University, which was very aggressive back then on the remote learning scene, even though it is a lot more common now. She got her undergraduate degree that way and then dived back into the masters program [Bace2006].

Timing worked out so that she finally finished her undergraduate work and hit the job market during a down time in 1982. She had gotten married in late 1981 and moved to Baltimore. Right before she got married, her mother-in-law took ill. In agreement with her in-laws, Bace went to school during the day, and took care of her mother-in-law at night. She stayed kind of out of the job market, just noodling around with consulting work here and there. She ended up with a happenstance connection to a firm located near her in a post where she ran the IT shop. She worked with them while they were doing a system change-over. It was a civil engineering job so she knew their applications pretty well and helped them with their outsourcing business.

Somewhere in there she read a magazine ad for the NSA who was recruiting like crazy. She sent them her resume, and just for jollies, sent them her husband's as well. Right about the time she had settled into her new job, the NSA came calling. They ended up hiring her husband before her and she continued working with her small firm. The NSA finally called her and offered her a job as well [Bace2006].

She talked to her husband that evening about how she did not think she wanted to leave her "Nirvana" job not four miles from her house; she liked both the job and the organization. But her husband said, "No, no, you've got to come to work for the NSA." Bace asked him to tell her three good reasons why she should. He pointed out that the commute was long, so they would actually get to spend some time together every day. He also brought up those guys that they remembered from academia who may have mismatched shoes on, or who think so hard walking down the hall that they run into the wall as examples of the types of brilliant co-workers they would get to rub shoulders with.

"The NSA is crawling with people like that. You were born to work here." So she went through the rest of the interviews and the NSA ended up hiring her, too [Bace2006].

The career values for Bace at that point made her want to work at a place where there was educational support, and the government is well recognized for supporting those with plans for graduate school. And she knew at that point that she did want to work up that part of the career chain. Bace also enjoyed being in a situation where folks were celebrating for having strong family values. A lot of it was classic goals [Bace2006].

Bace's current career objectives are obviously quite different. Things change over life. Bace works more and more with women who are coming through the career path at the executive level. It is the most rewarding work she does. She has ended up with the handle of "mom" or "den mother" [Thie2002]. Bace mentors for an organization called The Executive Women's Forum which functions as a portal through which young women figure out what they want to do in the information security/senior executive business field. And she loves it! It has involved into a passion over the years. This group consists of basically directors-level and above. They draw from academia, corporate, and government. It is a truly amazing group: professors, directors of large organizations, the heads of the big banks of New York. CSO magazine sponsors the group. Bace enjoys turning the whole notion of women in technical organizations on its head. Male colleagues understand that the women in this group have got something special that is actually of great value. A lot of Bace's male colleagues say how envious they are that women have something that they do not have and how it represents a leg up for them on the professional scene [Bace2006].

To a certain extent, Bace thinks it is important to be comfortable flexing with what gets thrown in an individual's direction. In terms of life, a lot of times a person might end up in a scenario or role that one would never have dreamt of for oneself. There is a place for being tenacious in general about life goals, but if one overdoes that, that person might be missing out on a better scene, a better set of possibilities [Bace2006].

In Rebecca Bace's case, computer security as it stands now really did not exist when she was a child. The computer barely existed when she was a child, at least not in a way that was visible to her. The availability and access to what is going on is also an issue. There is a lot to be said for understanding that flexibility is not necessarily the bad news that folks in career counseling would have students believe [Bace2006].

Rebecca Bace would agree that the history of intrusion detection has been shaped by female influence. When looking at the history of fields of science (specifically those pioneered back in the mechanical revolution, and especially computer science) it tends to become a traversal through the hall of male greats. Computer science seems to be inundated by male influence, even today. Recent survey results indicate that while women in science are gaining more doctoral degrees yearly, in the field of computer science there has actually been a decrease since the eighties in the number of women earning doctorates [Mosk2002].

Henry Louis Mencken once said, "For every complex problem there is a simple solution… and it is wrong." It is a good guess that the same euphemism can be applied to discerning the reason women seem to be drawn to the field of intrusion detection, the cross-roads of two traditionally male disciplines: security and computing. There are many influencing factors that could have brought about this increase of female interest in a traditionally male science. One factor could be the influence of the government since WWII when the draft moved a large percentage of men out of the country and out of the computer workforce. The government began recruiting women to step up and fill the gap, and women were hired to calculate missile trajectories and program ENIAC [ECP2006].

Rebecca Bace is putting her money on another aspect. And that is the fact that in computing, security in particular requires a different set of skills than classic computing. And frankly she thinks women do a better job with handling those aspects, and part of that is the matter of (her colleagues roll their eyes) if you break down the roles in computing, security folks end up being the physicians, or the health care providers. That requires a different set of skills, and a more integrated set of skills, than pure computing. Pure computing can accommodate those folks who end up doing a lot of the heavy lifting, those who may not be so verbal or socially adept, in a lot of cases. These types are bright beyond belief (intellectual prowess has never been an issue with them) and may be clashing in the authoritative working environment. Typically programmers are quite gifted but also very free spirits. A whole generation of management domain contributions focused on how to manage free spirits. These stereotyped free spirited programmers are sort of a cultural hallmark of computing [Bace2006].

With security, one does not get that luxury. The security consultant ends up dealing with folks who in most cases are grumpy. In the information age, they are feeling violated. One has to deal with them in a situation where clean-up after a catastrophe is going on, and people react in the same way they do as in other situations where they have been personally violated. In those situations it does not hurt to be a little more socially adept. In the end, it is how one deals with the carbon unit that determines whether there is success or not. There are a lot of things that traditionally this culture focuses on women and thus they are easier to do, and this includes the caretaking aspect needed for security and intrusion detection [Bace2006].

A lot of it rises from a dry academic research model of life where one has to strip all these aspects away and focus on the goods. There are a whole world of operators out there, but when the rubber hits the road in the operational sense, things are messier than they are ever going to be. The job of an academic researcher is to strip all that away as collateral. The issue with that is, in the operational world, what is viewed as collateral and pure research is actually the value proposition, which is what consumers are willing to pay money to resolve. In those situations it is not only a matter of a more comfortable marketplace; there really are compelling reasons that women do better in these things [Bace2006].

Bace has this to say about Denning, “She’s a really brilliant person. It helps to be working with a lot, and believe me, she works with a lot. Don’t let that shy and unassuming demeanor befuddle you. She’s a truly, truly brilliant person.” And as to the difference in interview styles, Bace confides, “She’s a lot better disciplined than I am. I’m much more a free spirit, much more like a guy in some aspects. I hate management etc… part of Dorothy’s greatness in this scene is she’s perhaps the most disciplined person I know. And there’s a lot to be said for that. She’s done extraordinary work in the area and deserves points and every bit of credit. I worry that she is so quiet and unassuming a personality that folks tend to downplay her contributions which have just been extraordinary [Bace2006].”

Bace considers her greatest contribution to be her mentoring work she does for the Executive Women’s Forum. On that she says, “I get to be mom, which is great fun. I get to be the keeper of the rolodex for the community.” It is a good match for a southern girl of

Japanese heritage. One of the upsides of growing up in the south is that she learned how to put together communities. Instead of that being something that is considered to be a questionable value, Bace thinks that in this day and age, connectivity is absolutely critical. She is not sure it is realistic to expect everybody to have those skills, but somebody on the team has to. It works out beautifully for the sort of things Bace does right now. She gets to keep the rolodex. She gets to make sure that when new thoughts come on the scene, they are not repeating themselves. She makes sure that new folks coming in get decent mentoring when it is available. Those things put wind beneath another person's wings, and it makes it a lot easier for a new person to accomplish her potential [Bace2006].

One key insight Bace offers is best understood from the perspective that everyone is still *at-point* in the life-cycle of intrusion detection. IT in general is still a pretty immature and unformed discipline, as the technical disciplines go. Something as relatively trivial and single-use as automotive transportation has taken the better part of a century to actually become real such that the fundamental things such as safety and so forth can be understood. Then having some sort of understanding how relatively immature IT is at this point is also appropriate. In the broader sense, it helps academics figure out what a reasonable expectation for them in terms of contributions are, but it is also helpful in terms of laying down some sort of reasonable impact characterization on the things that *do* get done [Bace2006].

A second insight was the importance of career flexibility. There is a lot of encouragement not to be flexible. A lot of the formal preparation is laid on you by the academic area, but because this is still by and large an immature area, it is important to remember that particularly when one is in situations where one may be struggling to make sense of something that simply does not make sense, it can be helpful to say, “Well, I’m willing to write that off to the immaturity of the area.” That is a situation where there is not a lot of encouragement to do that, but it is an important thing to remember in the big picture [Bace2006].

Raven Alder

Raven Alder got into intrusion detection by pure accident. As a graduate student, one of her teaching assistant responsibilities was to maintain the class mailing list and website. So essentially, she came out of graduate school with basic skills in HTML and UNIX administration. Due to her technical skills, she was hired as a network engineer in 2002 even though (like a true Renaissance woman) she held degrees in multiple disciplines. Although Alder had just started the job, her company already had a complex network in place. One of the router’s she was responsible for had crashed, and she had to figure out how that had happened. They had not updated their operating systems in several years, so one of the first things Alder did was to get a new operating system up and running for these routers. As part of the assessment she was doing beforehand, one of the routers got rebooted but did not come back up! This was because the operating system had been replaced with an MP3 of a Weird Al Yankovic song. It was someone’s idea of a joke [Alde2006].

Alder had to figure out how that happened, and that was the event that got her into security. The intersection between network engineering and security continues to be an interest of Alder’s, so much so that most of what she has done is backbone/network related [Alde2006].

Alder actively pioneers a lot of the backbone security aspects of her job, although there are certainly other people that are pushing the boundaries of that field. The theoretical research in security is a couple steps ahead of what is commonly put into practice, which is also true for most fields. But there are a lot of cases where people are not even doing the simple things that are known to be best practice because they do not believe the threats are real [Alde2006].

ZDNet Australia released an article in 2004[Gray2004] profiling five famous "hackers" putting their skills to good use. Raven Alder was chosen as the first candidate. The article got put up on Slashdot, and as Alder kept up with the posts, she felt really shocked in an unpleasant way by some of the comments responding to the article. "The immediate response was let's be really, really sexist about the girl [Slas2004]."

The Cisco threat of summer 2005 was the most tumultuous period of Raven Alder's career to date. Effectively, there was a remote boot exploit demonstrated in a backbone router. The exploit basically proved a point that had been long debated but no one had ever shown that in practice it was possible. Then in the summer of 2005, Michael Lynn came along and demonstrated that "Yes it is possible, I've done it [Ever2005]."

Cisco responded by trying to cover up the fact that this had ever happened. They censored the proceedings of the conference where the research was released and resorted to all sorts of means to prevent his research from being released. They slapped him with a restraining order so he could not disclose his research. Alder was giving a talk at a DefCon security event (in fact, she has the distinction of being the first woman to deliver a technical presentation at the famed DefCon hacker conference in Las Vegas) on how to further protect a backbone and how to make sure that routers and the network setup is secure as possible. Right before her talk, Lynn revolutionized her field by saying “Hey, you know that theoretical vulnerability? Well, look! I’ve done it!” So Alder ended up speaking about that and Cisco’s response which was actually a really big mistake on their part. A major vender like that with infrastructure everywhere should not be seen covering up and hiding the evidence of a major security problem. That kind of behavior does not inspire confidence in customers. Alder disclosed that a better response for Cisco would have been to admit it had been confronted with ground breaking new research, and then to respond quickly and appropriately by making new patches and releasing them. In this case, Cisco would have done much better projecting an aura of confidence and helpfulness, “‘Please call our support center if you need any help with this process. We are here for you.’ That would have been so much better [Alde2006].”

Because Cisco went through considerable efforts to suppress Lynn’s conclusions, Alder went through considerable effort to make sure they got out. “They weren’t very fond of me for that,” she remembers [Alde2006].

One key insight Alder offers comes from her experiences dealing with companies who have definite proven vulnerabilities and her attempt to get them to take fixes. The trick to presenting research to management, in order to get them to embrace change, is that it helps greatly to speak to what interests them. When persuading a major corporation to install a new kind of firewall, do not say, “Oh, this is really, really cool and awesome, you should totally get this.” No, you have to give them a reason to change. One of the things that a lot of brilliant technical people fail to do is to make a business case for things that need to be done to those people in management who may not understand the technical benefits and who might not embrace things that are technically correct because they may not understand why they are important. Ironically part of working in the field of intrusion detection is an entirely non-technical aspect that is just how to present work in such a way that it makes sense to those who need to act on it or implement it [Alde2006].

Conclusion

In conclusion, the walk from the inception of intrusion detection to where it is today has been shaped by several key female influences. In the beginning, Dorothy Denning was the initial trailblazer who pioneered real-time audit trail analysis for the first intrusion detection systems. Women like Rebecca Bace helped shape commercial intrusion detection into what it is today, and women such as Raven Alder continue doing their bit to move the field of intrusion detection forward through research.

Intrusion detection, and especially intrusion prevention, is a largely defensive maneuver, and as seen in history, women are often the ones left to guard house and home as the men go off to fight wars. Indeed, much of what constitutes good security practices are the meticulous setting up and fine-tuning of settings, practices which are often left undone through carelessness or ignorance.

By studying the influence of female figures on the history and development of intrusion detection, one is able to gain a sense of how women have contributed to the field's "excellence" (the current buzzword for diversity by the corporate workspace). By understanding how a broader range of perspectives has contributed to the excellence of intrusion detection systems today, this becomes a case study which provides concrete and long-term repercussions to how humanity may wish to approach gender balance in any area of life or specifically the areas of male-dominated science.

This section has focused on the work and wisdom of three key female figures in network security. Their insights capture the overarching mentality permeating the security field today. Denning demonstrated that using AI techniques in the field of intrusion detection can have profound and long-lasting results. In fact, the complete incorporation (if such a thing exists or is possible) of AI into intrusion detection is still not fully realized. There are still many aspects of intrusion detection that might be improved upon by using AI techniques. Bace showed that it is important to be comfortable flexing with what gets thrown in one's direction. In addition, she reminded the research community that as a science, intrusion detection is still "at point" in its lifecycle, and therefore one must be patient with setbacks and willing to embrace new routes that may lead to something unexpectedly better. Alder emphasized the need to present research results in an understandable (almost empathic) way to maximize the impact of breakthrough research or even to bring about change. It was through exposure to these women's experiences, contributions, and advice that the idea for a genetic programming solution to TCPMissing arose.

Current Breakthroughs in Intrusion Detection

After intrusion detection started rolling, it began to resemble a snow ball rolling down a mountain. It kept gathering more and more to itself, becoming larger as it picked up speed. In the twenty-first century, intrusion detection systems encompass levels of complexity sometimes staggering. For instance, recent work in intrusion detection has embraced various aspects of computing such as storage management and distributed computing to broaden and strengthen the abilities of intrusion detection systems.

In 2003, a storage-based intrusion detection system was set up on a machine's file server. This gave it distinct advantages when it came to monitoring activity and to maintaining compromise independence. Compared to a network-based intrusion detection system, one that has to track all the packets coming into the system, a storage-based model is much more efficient [Penn2003].

Having an intrusion detection system set up on the storage interface means that any changes to persistent data will be seen, and many intruder actions are themed around some sort of change to persistent data, be it manipulating system utilities to adding backdoors, tampering with audit log files to eliminate evidence, or resetting file attributes to hide changes. However, with this type of detection system, if the intruder does not attempt to tamper with the file system, then his actions may go undetected, but any actions taken will not be persistent across reboot. This is an argument for regular restarts of the machine [Penn2003].

A storage-based intrusion detection system is more efficient than a network-based intrusion detection system because it consumes fewer resources. It does not have to check the contents of every single packet because only the actions which attempt to modify the file system are pertinent. However, there is a trade-off in the amount of rules to check and the amount of resources used by the file server. No one would want to use the storage-based model if it significantly slowed down their system, especially when there was no suspicious activity going on. When tests were done on the storage-based implementation, they found that as long as no rules matched, the system performed the same for 0 rules or 1000 rules, which is an adequate starting point for good efficiency [Penn2003].

Another attractive feature is the compromise independence of the storage-based model. Host-based and especially network-based models are both subject to compromise. Due to the escalatory nature of information warfare, storage-based intrusion detection systems may be cracked someday. However, their very nature makes this a difficult proposition. The only way to access the storage interface is either through a special physical terminal or based on tunneling cryptography [Penn2003].

Another intrusion detection system that wins efficiency points is the distributed intrusion detection system which uses a Dimension-based Classification Algorithm to balance the load among processors [Shen2003]. The appealing aspect here is that multiple processors handling a heavy network load get the processing done faster because many processors have more resources to contribute to balancing the load (hence more efficient). Therefore, the packets are divided up and sent to different processors to be analyzed. However, the packets in one attack stream are interrelated to one another, and as such, should all be processed together on one machine to avoid inter-process communication, which is expensive. This is where the Dimension-based Classification Algorithm comes into play. Based on the algorithm, all the packets that should belong in one stream together are sent to one processor to be analyzed together [Shen2003].

In conclusion, IDS advances have evolved intrusion detection techniques to beyond the point of just explicitly checking packets for known attack patterns, although this still exists as a valid approach. It is common today to find far more subtle approaches to combat increasingly sneakier attacks.

Considering the complexity of current intrusion detection systems, it is important for IDS researchers to focus on the building block approach to research. Whereas the field of network security might be represented as a building, the wall of intrusion detection is made up of many well-placed, solid bricks. This thesis aims to add another brick to that wall in the form of a genetic programming solution to TCPMissing.

CHAPTER 2

AN ANALYTICAL INTRUSION DETECTION COMPONENT

The basic premise behind real-time intrusion detection is to analyze packets for malicious patterns as they come off the wire. In addition to performing this check as the first line of defense, it is also common to store network traffic in files for possible later analysis.

Files containing network traffic information essentially contain all the packets passing through the network in a certain time frame. These files can be stored either as binary or as text files, and they are usually referred to as “dump” or “trace” files.

As Internet speeds increase, it has become possible for the application performing the capture of the packets to have trouble keeping up. When that happens, certain packets may go missing from the dump file because the system was not fast enough to capture, process, or store all the packets. This is opposed to packets that may be dropped at the network level. Those packets require retransmission whereas missing packets do not.

Although packet loss information is generally computed and reported to the capture program, this information does not get stored and distributed with the file. An important tool in traffic analysis would be an application that could take an input file and determine the average loss rate caused by the capture program.

This information is vital to know in a post-attack atmosphere. It stands to reason that an analysis of the attacker's actions, methodology, and commands might yield valuable information as to the identity or motive of the attacker. In addition, understanding this information will help the defenders prepare the system for a possible repeat performance of the attack. Analysis of the attack is critical, but performing analysis on an incomplete trace file might lead to frustration and confusion, or worse, misinformation.

What seems to be required is an algorithm to run through the packets of a file, and while this algorithm would not be able to predict the exact contents of any given missing packet, at the very least it should be able to calculate the correct percentage of total missing packets. It turns out that the rules of TCP make satisfying this criterion feasible.

TCP as a Reliable Data Transfer Service

The theory behind determining missing packets is based on the rules of the Transmission Control Protocol (TCP). TCP is a reliable data transfer service in that it provides retransmission of lost data, provisions for out-of-order data, and even stipulations for how to handle duplicate data [RFC71981].

Every TCP connection begins with a three-way handshake, the hallmark of reliable data transfer. This special "handshake" coordinates the initial sequence number of both the source and destination host. The flag that marks these initial special packets is the SYN flag, to signify the process of synchronizing the sequence numbers. The sequence number used in conjunction with the acknowledgment number is what makes the TCP magic happen [RFC71981].

The sequence number is used by the source host to keep track of the amount of information already sent. The acknowledgment number keeps track of the amount of data already received by the destination host. If the source host does not receive an acknowledgement for the data sent within a certain amount of time, the source host assumes that packet was lost, dropped or mutilated by the network, and retransmits the packet. In some cases, it is possible for the acknowledgement to arrive right after this retransmission. This signifies duplicate data has been sent, but does not cause a problem because TCP is able to discard duplicate packets on the destination side when it is reconstructing the data flow and then only acknowledge the new data received [RFC71981].

One way to determine if a packet has been dropped by the network is to see if retransmission occurs after a timeout period starting after the time that the packet was first expected to arrive has already passed.

Another tricky scenario to handle is late packets, or packets that arrive out of order. Although TCP reorders out-of-order packets, it is possible to take the trace at an intermediary point (such as a router) where the packets are at the mercy of network conditions. It would be wrong to immediately classify a packet as dropped or missing if it does not immediately show up in the network trace in its proper spot. It could be late, and TCP allows for this by reconstructing the data flow in its proper order at the destination side [RFC71981]. Initially, an attempt was made to keep TCPMissing generalized enough to be able to run on a live stream of traffic as well as an input file. This made handling this scenario a bit trickier. However, the problem becomes moot once live traces are discarded because the simple solution is to reorder the packets in the file before computation.

Finally, in order to finish off the connection after all the data has been sent, a special flag is used to bring about the end. This flag is called “FIN” to signify that the source has finished transmitting all the data it needs to send for this session [RFC71981].

Despite the many complications that arise from dealing with a complex protocol for reliable data transfer, it is possible to develop an algorithm to determine missing packets based on the rules of TCP. This is done based on the parsing of packets in the trace and looking at the fields in the packet TCP header. The specific fields which are of interest to the determining algorithm are source IP address, destination IP address, source port number, destination port number, acknowledgment number, sequence number, and certain flags such as SYN or FIN.

The basic concept is to arrange the packets into their proper flows (connections) and from that standpoint begin reconstructing the data stream. If a point is reached in the stream where there is a discontinuity in the sequence numbers of the packets, a determination has to be made. The packet could be late. To determine this, keep reading in the stream. Or a packet could be dropped or timeout. To determine this, look for duplicate packets or retransmission. Or a packet could just be missing from the trace and this is what we attempt to determine by looking at the immediately surrounding packets to see if the data stream acts as normal.

Implementation of TCPMissing

TCPMissing is a program that evaluates a dump file to determine the number of missing packets caused by the application because the system it is running on is not fast enough to capture, process, or store all the packets. This is opposed to packets that may be dropped at the network level. These packets require retransmission whereas missing packets do not. Throughout this section the distinction is made between these two types of packets by always referring to the first type of packet not captured by the application as a "missing" packet. It is important to realize that this packet belongs in the trace. It was on the wire with the other packets. It presumably eventually arrives at its destination.

The second type of packet that never reaches its destination is a "dropped" or "lost" packet. Packets are lost due to congestion on the network. When two or more hosts attempt to transmit at the same time, both of their messages become garbled. TCP can attempt to recover from congestion using either the slow-start method or fast retransmit. Even though a message can safely make it out of the host network, it may have to take many paths to make it across the Internet to its destination network. It is during any of these segments that congestion may occur and the packet may be lost. TCP recovers from lost packets by providing retransmission if a timeout occurs and no acknowledgment is received from the destination.

Two java files were needed to write this program. *Tcpmissing.java* held the file parsing and flow assignment code. *Flow.java* was where most of the flow processing and analysis took place. TCPMissing was coded in Java to take advantage of the JPCAP library. JPCAP is the Java overlay to LIBPCAP, which is the format that all dump data files are saved in [Cars2003]. It was necessary to download the appropriate Java library extensions since JPCAP is not part of the standard Java library.

The following command prompt line compiles both files necessary to run TCPMissing: *"javac Tcpmissing.java Flow.java."* Similarly, to run the TCPMissing program, execute the following command: *"java Tcpmissing [filename]."*

Command Line Parameters

Required:

args[0] - trace data file name

Optional:

args[1] - keyword "where" : which packet to eliminate, optional with args[3] and args[4]

args[2] - integer value denoting location of which packet to remove

args[3] - keyword "num" : how many packets to remove, optional with args[1] and args[2]

args[4] - integer value indicating how many packets to remove

args[5] - keyword "stats" : prints out results and statistics

If the keyword "where" is found in the command line, an integer is expected to follow it as the value for variable "luckyPacket." Likewise for the keyword "num;" the program will look for an integer value in the next args[] to initialize "miss."

The keyword "stats" is desirable for individual program executions. However when batch files are being run, the omission of this keyword will bypass the onscreen results display, thus saving valuable time when processing multiple executions to generate valuable statistics. Because removing a packet from one position verses another might affect results depending on whether it hit upon an uncovered special case or not, it was necessary to perform multiple runs through the data removing the packets from different positions and then average the results.

The results can still be viewed by accessing the output file generated and named according to the "num" parameter. The file name generated will be of the following format:

Filename format: (miss.toString() + miss)

Example: If two packets are removed from the trace, miss equals two and filename is "2miss."

Batch Scripts

In order to generate data for a test case, it was desirable to use a file which had already been pre-determined to contain only one missing packet ("*real1b*"), and this only due to a flow being recorded from the middle of the stream instead of catching it at the beginning. Thus the one missing packet it caught was legitimate. In order to test the TCPMissing code, this otherwise whole file was used as the starting point from where to semi-randomly remove different numbers of packets and then evaluate the performance of the program at predicting the corresponding number of missing packets.

Since *Tcpmissing* is a rather static class, writing another program to repeatedly create instances of the *Tcpmissing* object and iterate through it that way was not a workable option. The separate class in question would have also had trouble opening and closing multiple files. Instead the workable solution was to run the program once but call it multiple times from the command line using a simple yet elegant batch command:

for %a in (0 1 2 3 4 5 6 7 8 9) do command /c for %b in (0 1 2 3 4 5 6 7 8 9) do java Tcpmissing real1b where [%a%b] num [how many to remove]

The above command is a double *for* loop which runs *Tcpmissing* one hundred times in order to remove the packets from a variety of different semi-random places. Every time the program completes, the results are appended to the output file ([how many removed] + "miss") in the following format:

%a%b Tcpmissing.missing

The first parameter identifies the current iteration through the program. The second parameter is the number of calculated missing packets. Files saved this way can be opened and graphed later to see the mean result.

Since the resulting data is appended to the end of an existing open file (the appending flag is always set to true), re-computations should only be attempted after first clearing the file of all old data. This is done simply in the command line with the following command:

type " ">[filename]

Javadoc Generation

Although at present perhaps TCPMissing might be a bit too specific for general reuse, the decision was made to document the work in order make it available for perusal online. The following command created the html files in a javadoc directory:

javadoc Tcpmissing.java Flow.java –private –link

http://java.sun.com/j2se/1.4/docs/api -link

http://netresearch.ics.uci.edu/kfujii/JPCAP/doc/javadoc -d javadoc

The first link shown above ties the TCPMissing documentation to the main Java.Sun API homepage. The second link tied the TCPMissing documentation to the JPCAP classes used to implement the program. These JPCAP classes are not found on the main Java.Sun API homepage. The resulting documentation for the TCPMissing classes can be found in Appendix A.

Challenges

The first challenge arose from the decision to use JPCAP instead of manual parsing. Most work underway at UGA at the time relied on manual parsing of regular text files. Since no one else had led the way using JPCAP parsing of binary files, it was at first uncertain how to learn it, how to install it, or even how to acquire it. In fact, various versions of JPCAP exist on the web. One of the greatest confusions encountered resulted from the fact that documentation had been found for a different JPCAP package than the one that had been downloaded and installed. After things were back on the same page, everything began to make more sense and fall into order. The JPCAP package that had been downloaded was easy to learn, if not documented very well. This made the file parsing task achievable.

The actual packet parsing had been the second major concern. Once JPCAP became understandable, it was obvious that with a couple well-placed commands, one could go from a large binary file to its constituent packets easily and quickly. However, the problem then became how to allocate enough resources to handle large volumes of packets. This proved to be more difficult than first anticipated. It was postulated that normally it would be a simple matter to work within a moving window of packets and remove packets on-the-go. That way, the system would only need to keep track of a workable number of packets at any one time. With the programming language C, the keyword *delete* could have been used to help allocate and de-allocate memory. However, Java's automatic garbage collection and the non-existence of *delete* made this task a little more challenging. With large amounts of packets (more than around 6000), TCPMissing would terminate with an EXCEPTION_ACCESS_VIOLATION.

After many hours trying to decipher and debug this problem, it was decided to work around it by processing dump files at about 4000 packets at a time. This problem occurs because the java virtual machine runs out of memory processing extremely large files, especially those consisting of large numbers of incomplete flows. These incomplete flows represent a problem because they cannot be processed till the end of the file after it is certain there is no more data. Otherwise, a flow might be prematurely processed and cause errors down the road. So the program keeps waiting for the FIN flags to signify end of data so that it can begin processing the flow for missing packets. This means that more and more flows keep building up till the system runs out of memory.

This problem has a simple work-around. When executing the program on the command line, specify additional parameters to increase the size of memory the Java virtual machine has to work with:

```
java -cp . -Xms100m -Xmx300m Tcpmissing [filename]
```

Special Cases for Packet Analysis

There were many special cases to keep in mind when handling the packet analysis. In addition to treating FIN's and SYN's differently than normal packet transactions, it was also necessary to write separate code to allow for duplicate packets, out of order packets, and packets sent when the receiver's window shrinks to zero. It was falsely predicted that flow processing would be moderately easy once JPCAP was installed and parsing packets. It seemed like a straightforward task that only required a little time, effort, and clear-thinking. However, it was not, and there were many weeks spent pouring through output trying to reorganize the algorithms to account for many cases not previously anticipated.

The basic idea was to try to predict the other guy's next acknowledgment number based solely on the packets that had come before. This restriction was imposed because down-the-line it would make converting the program to analyze run-time traces easier. If the next packet did not have the predicted ACK number, either it was a special case, or some packet was missing from the trace.

Normal Case

In the usual case, the predicted ACK is generated by adding the data length onto the sequence number.

SYN Case

Even though a SYN packet does not carry any data, the acknowledgment that comes back must be incremented by one.

FIN Case

A fin packet may or may not carry data with it. It was possible to write code that handled the case with data by treating it similarly to a normal case and the case with no data by incrementing the sequence number by one, much like the SYN case.

Duplicates

Here the identification field became helpful. If two packets were found with the same identification number, then it was just a duplicate packet case and not a missing packet.

Delay

Sometimes packets can arrive out of order. Again, special code had to be written that would allow for this and not automatically increment the missing packet counter.

Window Zero

As part of TCP flow control, the receiver advertises a zero window when more processing time is required on an already full incoming buffer. In this case, the sequence number would equal one less than its predicted ACK number. And whenever this happens, the window size is set equal to zero.

Other Cases

There were several other cases that were not implemented. Those cases were determined to occur so rarely as to not seriously affect the results of preliminary testing of the system and so they were left undone to see if artificial intelligence techniques could deduce all the cases and outperform a (marginally) incomplete algorithm.

One example of a special case that was not implemented was if two packets are missing one right after another. The algorithm will be able to detect that a gap occurs, but will not be able to correctly determine the exact number of packets that should be filling that gap.

Results

Testing the program required the use of the tried-and-true data file in use since the beginning. When first starting to play with JPCAP and learn everything it could do, a file called "*real1b*" was used to figure out how to parse packets. There were two primary TCP flows in this file. They are plotted below.

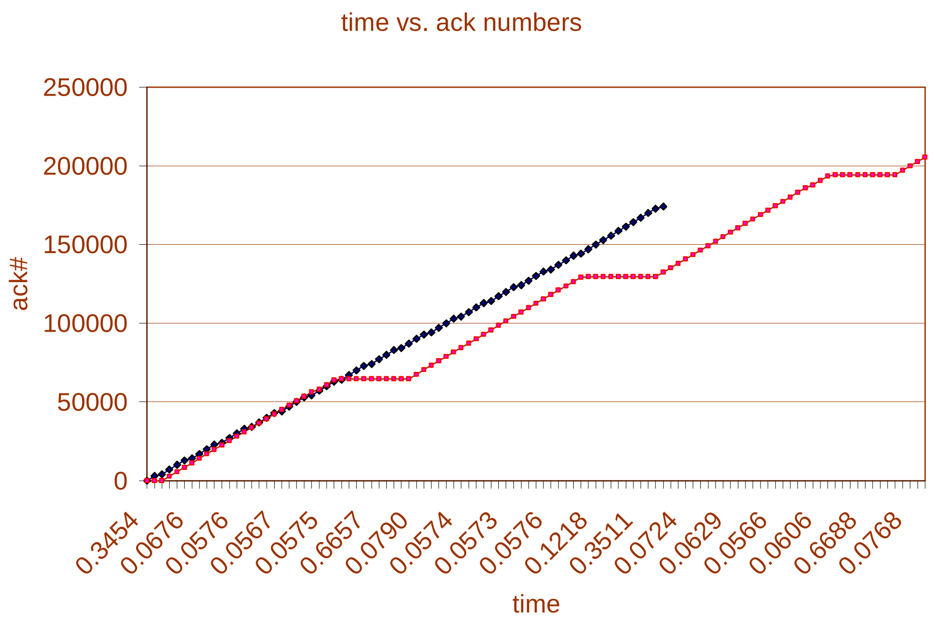


Figure 1: The depiction of two flows within the dump file "*real1b*"

The graph shown above depicts the data transfer progress of two flows. Each dot represents one packet. The dot is colored black or red depending on the respective flow it belongs to. The acknowledgement number for that packet determines the dot's placement on the y-axis. The x-axis is change in time. As time increases, the acknowledgement numbers increase for both of the flows, signifying the successful transmit and receipt of data.

One flow (depicted in black) experienced regular, steady data transfer. The other flow (depicted in red) displayed signs of heavy transfer followed by three distinct periods of "zero window" behavior where the dots follow a path parallel to the x-axis signifying no receipt of new data. As files were parsed and graphed, it was fascinating how often flows had the tendency to develop predictable patterns. In the following figures, the packets graphed are all acknowledgment packets. This was done to emphasis the importance of acknowledgment numbers in the determination of missing packets.

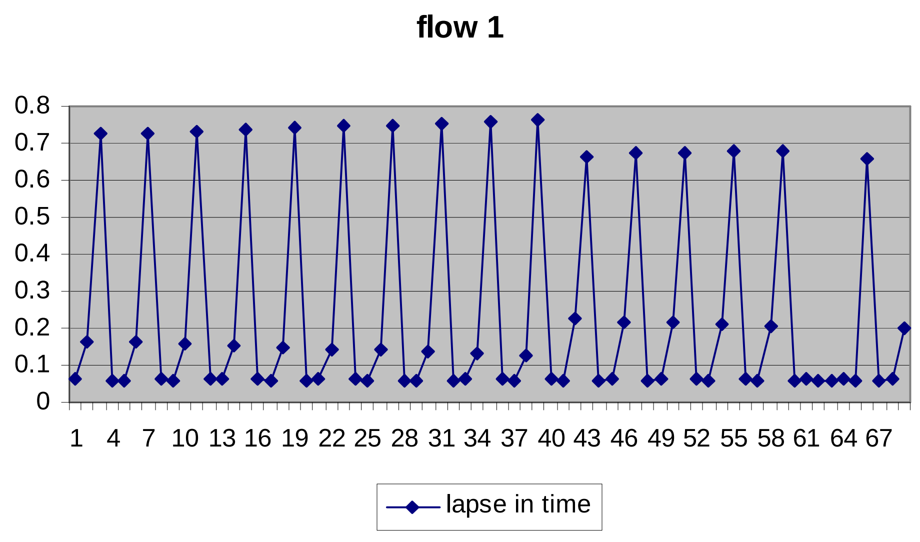


Figure 2: Representing flow 1 based upon the difference in the amount of time between each acknowledgment packet sent

Flow one is the steadily increasing (black) flow from the previous chart. This figure shows the temporal difference (delay) between all the packets from the receiving (destination) host in the transmission. The y-axis varies from zero seconds up to 0.8 seconds. The x-axis contains the packet number. So for example, packet number three was delayed almost 0.75 seconds after the last packet, but packets number four and five were sent less than 0.1 seconds after the last packet in the same flow.

This graph, seen in conjunction with the next, demonstrates the operation of delayed acknowledgements in TCP. When the recipient host receives a packet from the source host, it will wait a small period of time to see if another packet from the same host comes in and acknowledge both of the packets at once. This delayed acknowledgement behavior dramatically decreases wasted bandwidth, as acknowledgment packets for single direction transmissions do not contain any other data of their own. However, in order to avoid a timeout, the destination host only waits so long for the second packet to come. In this case, after about 0.7 seconds, it would go ahead and just acknowledge the first packet received.

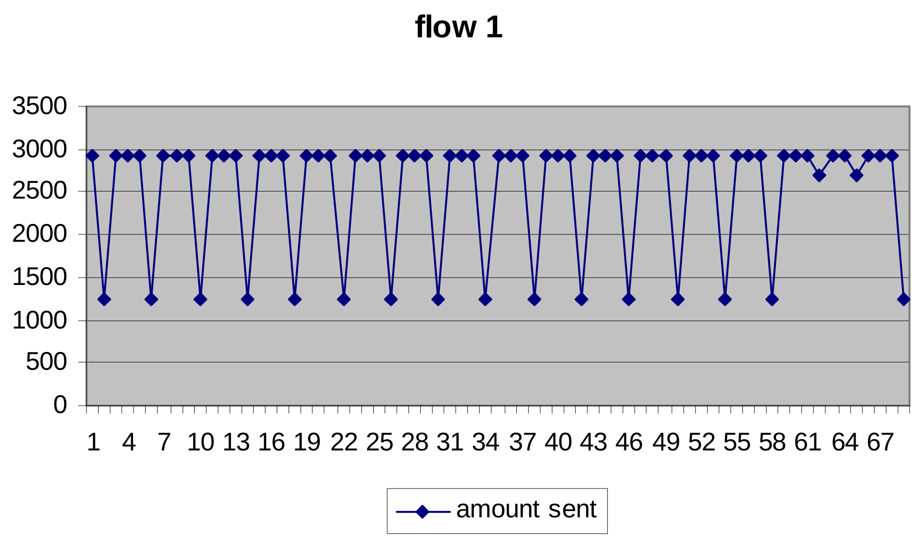


Figure 3: Representing flow 1 based upon the amount of information acknowledged by each destination host packet

The graph above makes this delayed acknowledgment business a little clearer. It is similar to the last graph in that it represents the acknowledgment packets from the destination host, only this time the y-axis is the amount of information acknowledged by every packet. Every three packets (for example packets three, four, and five) were acknowledging about 3000 bytes of data apiece. This is because the source host was consistently sending out packets of size 1460 bytes apiece. The destination host would wait till it received two of those packets adding up to 2920 bytes and then acknowledge that amount of information received. However, then the destination host would receive only one packet from the source and be kept waiting a while for the second. To avoid causing a timeout and retransmission by the source, the destination host does not want to wait too long to acknowledge packets, so it went ahead and acknowledged 1460 bytes received in packet six. Although this seems like bursty behavior, overall it provides for the steady transmission of data (no need for retransmission) that supports the slow-and-steady increase behavior displayed in the very first graph.

This flow analysis has been done on the file "*real1b*" so that it might become apparent how even simple looking flows can easily fall into complex patterns of which TCPMissing must be careful not to be deceived by. For instance, it is not enough to simply check for delayed acknowledgments for every two received packets. As seen, acknowledgements can also be sent after just one packet to avoid timeouts. This analysis of "*real1b*" helped in the design stage of the problem to avoid common pitfalls and establish special case scenarios of delayed acknowledgements verses immediate acknowledgments.

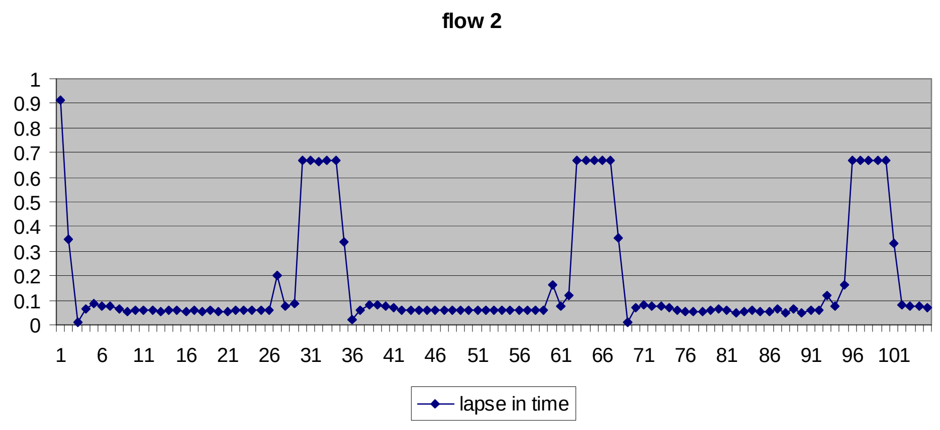


Figure 4: Representing flow 2 based upon the difference in the amount of time between each acknowledgment packet sent

Just as flow two levels off in three places in the first graph, it is apparent here where the window cuts to zero in three distinct places. It is in these places that plateau behavior occurs as long delays of about 0.7 seconds occur between each packet. Again, here a clear pattern is discernable as change in time is plotted along the y-axis and packet numbers are plotted along the x-axis.

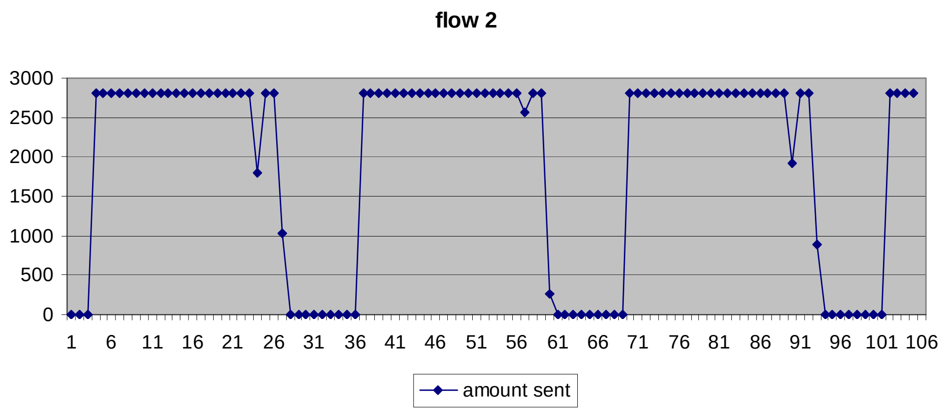


Figure 5: Representing flow 2 based upon the amount of information acknowledged in each consecutive destination host packet

It is perhaps even more blindingly apparent what is going on in Figure 5. Here packets are plotted according to the amount of information acknowledged by each one. Packets 0-3, 27-36, 61-70, and 94-100 all acknowledge 0 bytes from the sender. Because these empty packets are grouped together and correspond to the periods of delay (long wait between successive transmissions) shown in Figure 4, it points towards a zero window scenario where the sender must wait for permission from the receiver to increase packet size up from zero. This analysis of "*real1b*" helped in the design stage of the problem to establish the special case scenario of zero window behavior.

In order to test the TCPMissing algorithm, the file "*real1b*" continued to be used as the default test-case file. When running "*real1b*," the TCPMissing program finds one missing packet. This is correct because one flow starts in the middle (no SYNs) so the program counts the first packet in the interrupted flow as a missing packet case. This was taken as the base input and scripts were written to parse through “real1b” and remove one packet from a hundred different semi-random places and calculate the results for each iteration. Since this was done a hundred times and on different packet positions, presumably different scenarios would be touched upon every time triggering different branches of the algorithm. After completion of all the runs, the average could be computed to see that the resulting percentage predicted missing was pretty much what was expected. This was also done for two, four, and ten removed packets.

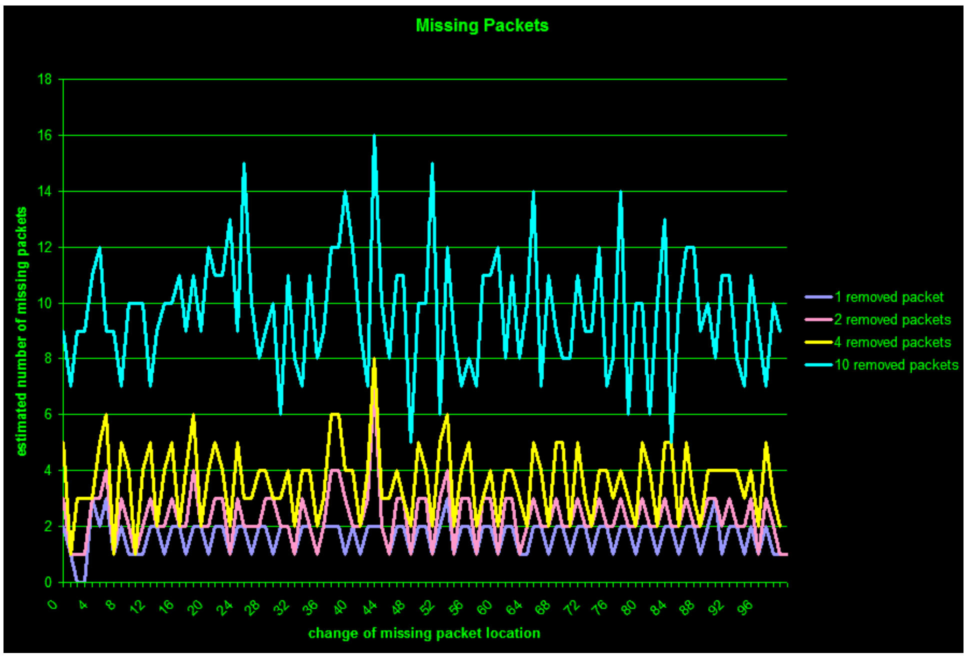


Figure 6: The average number of projected missing packets as computed by TCPMissing for 1, 2, 4, and 10 removed packets

As seen from Figure 6, the purple line on the bottom depicts the average results obtained from successively removing one packet from different locations in the file. The pink line depicts the average results for removing two packets from the file. The yellow line depicts the results of removing four packets. And the blue line depicts 10 removed packets.

Depending upon the different positions the packets could be removed from, the program tended to choke on packets removed from special cases scenarios not explicitly coded for in TCPMissing, and it was this type of behavior that prompted initial interest in having an artificial intelligence technique to learn to perform this type of computation and infer the special case scenarios.

Despite this unfortunate behavior for specific runs, averages of total runs tend to stick pretty closely to the actual number of removed packets.

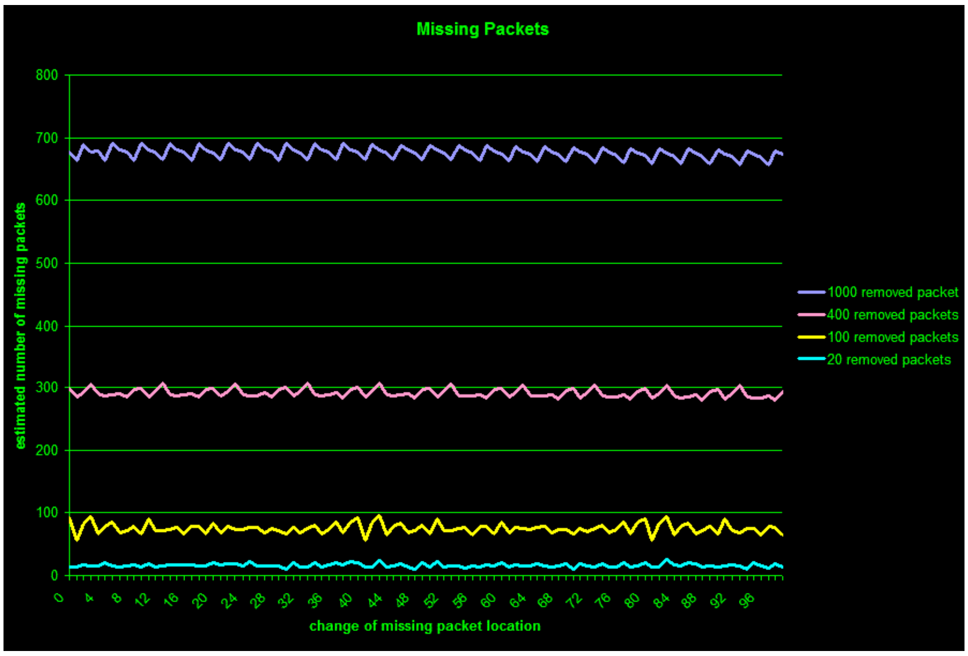


Figure 7: The average number of missing packets as computed by TCPMissing for 20, 100, 400, and 1000 removed packets

Attempting to remove more packets than around 5% skews the results and slowly the TCPMissing program falters and begins catching less and less missing packets. This behavior is shown in Figure 7. To a certain extent, it should be possible to make the TCPMissing program more robust, by adding more special case scenarios, and that should be able to increase the percentage it can compute before it falters. But it is important to recognize that eventually one must reach a limit because the TCPMissing program is dependent on packet information for results. It is only logical that with not enough packet information, it will not be able to compute results correctly. Eventually, with enough packet loss, even a person would have trouble reconstructing the flows. As more and more packets go missing, the less information TCPMissing has to work with to correctly carry on with computations.

Conclusions

The results shown above are promising. Because not every single existing special case was accounted for, there was a little variance in the outcome depending upon the location where the packet was removed from. However, the overall averages were consistent.

TCPMissing advances research in the field of intrusion detection by determining the number of missing packets in a file created by a traffic capture application. Even though TCPMissing is an important component of any network security system, it had not yet been implemented. This project does that. TCPMissing shows that predicting missing packets that are supposed to be in a trace can be done with reasonable accuracy.

It remains for future work to refine the techniques used by TCPMissing to yield greater accuracy over more specialized cases. In fact, a better solution than trying to predetermine every existing special case and code for each one would be to incorporate AI learning techniques into the system. In the next chapter, genetic programming is used as an alternate solution to the TCPMissing problem. The GP implementation ends up being faster to code, more accurate, and more efficient.

CHAPTER 3

AN INTELLIGENT ANALYTICAL COMPONENT

Today intrusion detection systems are complex and intricate. Many commercial ventures have grown out of the opportunity to tap into the market for security from an ever growing Internet threat. Even though many different efficient systems currently exist, the setup/configuration is often neglected by system administrators. Partially because of this reason, and the sometimes slow response of humans as compared to computers, artificial intelligence techniques have become popular in the intrusion detection domain.

Previous work has been done in combining the areas of intrusion detection systems with AI learning techniques as far back as 1986 when Dorothy Denning blazed trails with her development of a general model for a real-time intrusion detection expert system capable of detecting penetrations and other forms of computer abuse [Denn1987]. There was steady progress made in the intrusion detection field during the next twenty years. In 1992, Koral Ilgun based his thesis on a real-time intrusion detection system, an expert system that analyzed state transitions [Ilgu1992]. In the paper done by Jeremy Frank in 1994, he outlines the current and future approaches for incorporating learning techniques into intrusion detection systems [Fran1994]. In 1995, Sandeep Kumar completed his dissertation by studying pattern matching as a means to represent and detect intrusions [Kuma1995].

Artificial Intelligence Techniques Showcase

This section is a partial future work section masquerading as a survey of AI techniques. It presumptuously assumes that the industrious researcher might take any of the following techniques as a starting point for further exploration.

AI techniques are many and varied, and there are many valid ways to apply AI techniques to a problem. Usually one is chosen over another depending on the exact nature of the goal trying to be accomplished. This section will highlight a basic AI technique and then describe how it could have been used in the setting of the TCPMissing problem. The next section will focus on genetic programming and why it was chosen as the AI technique of choice for TCPMissing.

Artificial Neural Networks

A neural network consists of a set of highly interconnected nodes [Cann2000]. Depending upon the weights attached to each node, and the manner in which the network is trained, it will become consistent in responding with similar outputs to similar inputs. Essentially it is a black box that can be trained to respond in an “intelligent” manner [Cann1998].

Because of the learning abilities of neural networks, they have often been an attractive option for detecting intrusions [Deba1992], and they were an attractive option for implementing TCPMissing. The algorithm developed to determine missing packets in a

TCP trace file is exactly the type of black box problem that can be fed into a neural network. The theory is that a neural network could perform as well as or better than the algorithm. Based upon current research, no one has attempted to do TCPMissing with a neural network before.

Expert Systems

Expert systems also have the ability to continue learning new patterns [Dass2003]. Expert systems are useful in cases where the necessary knowledge can be determined and encoded into the system in such a way that encapsulates the knowledge of a similar human expert. Often during the development phase of the system, a corresponding human expert is brought in to help create the knowledge base.

An expert system would be an attractive option to pursue in regards to TCPMissing since the developer of a TCPMissing algorithm would have reached the pre-requisite expert-level of knowledge needed to encode an expert system. In addition, a TCPMissing expert system could learn new special case scenarios as it came across them.

Genetic Programming

The genetic programming approach empowers the researcher with the ability to solve difficult problems or achieve better human performance on “tricky” problems. In general, this is done by finding a best guess approximating solution. This is done based on the evolutionary techniques of representation, mutation, recombination, parent selection, and survivor selection [Eibe2003]. Genetic programming was chosen as the AI supplemental technique for TCPMissing because the algorithmic approach used in TCPMissing seemed slightly incomplete as it did not account for every possible special case scenario. One possible scenario was when two packets were missing in order, back-to-back.

It was proposed that a genetic programming approach would be able to extrapolate the subtle, if however rare, cases not explicitly checked for in the sequential algorithm. Throughout this chapter, the first implementation of TCPMissing is denoted as “the algorithm.” This second implementation which utilizes genetic programming is referred to hereon as the GP solution.

A Genetic Programming Implementation Using ECJ

The genetic programming solution was implemented using the Java-based Evolutionary Computation Research System (ECJ). ECJ contains helper classes that provide the basic evolutionary techniques of representation, mutation, recombination, and parent or survivor selection necessary to emulate artificial evolution [Eibe2003].

Despite Java’s reputation for slowness, it was again chosen as the high-level programming language of the GP solution. This was for several reasons. First, because the previous TCPMissing algorithm had also been written in Java, it was desirous to keep the project uniform by writing any later modules also in Java for easy incorporation into the mother program. Second, despite Java’s reputation for slow computation, Java code can be optimized to perform well [Davi2005], and third, the object-oriented approach better captures the core nature of TCPMissing. The separate fields in the packet are represented as separate nodes within each individual tree. Java, as an object-oriented language, defines these nodes as classes and this translates into straightforward code that is easily interpreted by anyone trying to analyze the code. Furthermore, Java libraries exist for basic JPCAP and ECJ functions which can easily be built upon, engendering and encouraging code reuse, which is always an important consideration to a code developer.

Perhaps most importantly, architecture-neutral Java is portable across multiple platforms and perfect for the heterogeneous network environment that is the Internet [Gosl1996].

System Simplification

Based on a proof-of-concept mentality, the purpose of implementing a genetic programming solution was to see if an ideal individual (or solution tree) could be found to successfully predict the number of packets missing from a trace. A good starting point was to ascertain that genetic programming could handle the simplest case. The simplest case was determined to be a file containing packets from one single specific port on one specific Internet host (or one direction of a single flow) with no duplicate packets, no out-of-order packets, and no timeouts. Window size of zero was taken into consideration since the TCP response to that scenario is imperceptibly similar to normal functionality, at least from a mathematical standpoint.

Varying packet length was taken into consideration since packet length can vary from the size of the header information (but no actual message content when window size shrinks to zero during congestion) to the maximum segment size when network traffic flows fast and freely. In between these two extremes are any possible intermediate lengths than can occur based on various possible scenarios. One such case occurs when the push flag is set. Usually, the sending program waits until the output buffer is full before transmitting. The push flag signifies that all current buffered information must be encapsulated and sent immediately instead of waiting for enough content to completely fill up the packet. This represents a core TCP behavior, so considering its importance and frequency, the genetic programming approach was trained on input data of varying packet size, and it was considered part of the simplest-case scenario.

To increase the functionality of the genetic programming solution to encompass all the rest of the special case scenarios, a conjunction of a variety of different techniques would be required. The first step would be to run a new dataset containing a larger representation of more TCP scenarios through the first proven simplest case model. Then any missing packets not detected by the ideal individual would be new targets for future genetic programming work. This process could be repeated as long as there are new scenarios to learn.

Node Selection

The first step in creating a genetic programming solution is deciding on the essential nodes. The more complex a problem is, the more factors the problem will have in its equation, and thus the more nodes it will contain. Nodes are used as operators or operands in the tree, and depending upon their placement in the tree, affect how the tree is evaluated. A tree is a representation of the solution equation. This method of tree representation using connected nodes enables the solution of the problem to be modeled mathematically where a traversal through the nodes of the tree will yield the equation.

In the framework of ECJ, this evaluation is done recursively. To start with, the evaluation function is called on the root or top parent node. In order to determine the value of the root node, it must evaluate all of its children first. Often, the children of the root node are parents of their own trees. Thus, control of execution continues traveling down the tree until a terminal (or leaf) node is reached with no children. Then values begin passing back up the tree until all branches have been evaluated and the root node can finally compute its value, too.

In most cases, variables are terminal nodes on the edge of the tree, so they can not get their values from their children, because they have none. Their values are the variable inputs for evaluation of the tree. They represent the data objects of the equation.

It is the operators, such as add, subtract, and multiply, which make up the function set and populate the inner nodes of the tree in order to act upon the variable nodes. These mathematical operators all take exactly two nodes as children, and are responsible for manipulating them in such a way as to determine what value gets passed back up the tree. In the case of add, the add node simply retrieves both values from its two children, adds those values together, and returns the result.

There can be more complex inner nodes as well. Consider, for example, conditional or switch nodes. A conditional node may have a variable number of children. Depending on the value contained inside a conditional node, it selects which branch of the tree to follow, or which child's value to return. For example, in TCPMissing, the flag field of a packet header could be modeled as a conditional node. It is possible for the flag field to contain several possible values: 'SYN' to represent the synchronization bit and indicate sequence numbers needed to be initially synchronized, 'FIN' to represent the finish bit and indicate that the sender has reached the end of its byte stream, 'PSH' to represent the push bit and request the immediate transfer of data, 'RST' to represent the reset bit and indicate the connection must be reset, ‘URG’ to represent the urgent bit and indicate that the urgent pointer field is valid, or simply ‘.’ to represent no currently set flag. To adequately model each of these separate cases, the flag node would have five children. The children could be terminal nodes or in fact just the beginning of longer branches of the tree. It does not matter. The important point is that depending upon the value inside the flag node, only one branch of the tree is chosen for execution. If the flag bit was set to ‘SYN’, the first child would be chosen (the one corresponding to the ‘SYN’ choice) and that would be the branch of the tree modeling the case where the packet is a synchronization packet, which would be a different case than if it were a ‘PSH’ packet. Although a switch node on the flag field will probably prove useful in an exhaustive implementation of this problem, for the purposes of a proof of concept this node was eventually left out of the tree node final selection. This is because the final nodes modeled the most basic of cases, and modeling all flag selection cases starts to deal with TCP behavior complexities that fall out of the bounds of the scope of the current problem.

Another possible switch node based on packet header information is the source verses destination address. Depending on whether the current packet under consideration is from the source or the destination might signify a difference in how the packet is processed. Again, depending on whether the source or destination bit was set in this switch node would determine whether the source or destination branch of the tree was traversed. Eventually, when the simplest case was derived, this node was unneeded as well, since in the simplest case, one deals with one side of a single connection.

In the end, there were ten final nodes selected for evaluation. Seven nodes make up the data objects set. *Length* was one of them, and contained a value of type double that represented the length of the data transmitted in the packet. *Length* was a terminal node, meaning it had no children. *Seq* was another important terminal node used to represent the sequence number of the current packet under investigation. It contained one value of type double. *Ack* also contained a double value that represented the acknowledgment number of the current packet under consideration. *Ack.java* is included in Apendix B as a representative example of a *GPNode* customization for the TCPMissing problem.

Another terminal node was *RegERC*, the regular ephemeral random constant. *RegERC* contained a value of type double that was initialized to a random number by the program. This node was helpful in generating random constants. Having a constant in an equation may mean success by providing a better fitness value than that gotten from just variables and operators.

Memory nodes were utilized to keep track of the packet that had been seen before, so that it could "remember" where it was in the current stream of data, and forget everything else. *M0Ack* was a terminal node containing a double value representing the memory node's acknowledgment number. In this case, *M0Ack* was always set to the value of *Ack* from the immediately previous packet.

Similarly, *M0Seq* was a terminal node containing a double value representing the memory node's sequence number. Used in conjunction with the memory node's length, a double value stored in terminal node *M0Length*, these three values from the memory node should be sufficient for the base case discrimination of missing packets.

In addition to those seven terminal nodes, there were three internal nodes to help with the processing, and these made up the function set. *Add* took its two children's values and added them together and returned the resulting double value. S*u*b subtracted the second child’s value from the first and returned a double value representing the results. *Mul* also depended upon the values of its two children; in this case it would return the result of multiplying the two children together as a double value.

The ten nodes discussed above were determined to be sufficient for testing the simplest case scenario, and in fact, later it will be seen that this is true. This simply means that arranging a subset of the seven variables and three operators into some sort of equation will yield the proper answer for whether a packet has gone missing or not when evaluated on each line of input from the dataset.

Creating Input

The input file for ECJ differed slightly from the input file used by the previous TCPMissing algorithm. The TCPMissing algorithm opened up a dump file, and using JPCAP, parsed out packets and represented them as objects in memory. The same approach could have been taken with ECJ. Indeed, the two programs could be merged to perform the same calculations on one input file in tandem. However, for simplicity and clarity, a step-by-step approach was taken, at least for this initial proof of concept. The end result is that there are several modules involved in the whole process.

The first module used was the Windump facility. This is a freely available program used in the capture and processing of network traffic. The same exact data file was used for ECJ as was used in testing the TCPMissing algorithm, although with three-fourths of the data trimmed out so that only the traffic seen by one-host in one connection would be presented to the tree. Instead of reading the binary format of the file, a preprocessing step was taken to simplify the input file. Running from the command line, the Windump utility was used to parse out only the packets seen by one host in one connection with the following command: *windump -ntttvv -r real1b host 128.192.101.108 and port 4902 >> data/abso2.* The same command could have been programmed in Java and taken place inside ECJ.

The flag *–tt* requests that the format of the timestamp be the difference of time in microseconds from the time of the last packet received before it. The flag *–vv* requests that verbose output be activated, resulting in fields such as ID number and time-to-live to also be printed out. All these values were determined to be valid information possibly needed for the successful determination of packets from special cases. However, they were eventually unneeded in the simplest case scenario.

The resulting file, in this case *abso2*, contained one side of one connection. Next, it was necessary to parse out some information and add another column to the file containing information about the state of the file. File state information represents the number of missing packets and is necessary to evaluate the fitness of trees in the genetic programming solution. The state information was represented as an integer, either zero or one. If a packet was removed from the immediately proceeding line, the value in the state column was set to one. Otherwise if no packet was removed from the proceeding line, the state was left at zero. These values of either one or zero, depending upon whether the successive packet was removed or not, proved the basis for the fitness function.

The module that took care of this functionality was *Gen2.java*, taking as input a file generated by Windump and outputting a file ready for processing by ECJ. Again, this module is entirely capable of being fully incorporated in the ECJ main program, but for clarity’s sake was left to stand alone to demonstrate the different steps of input preparation.

The following command was used to compile the program: *javac -classpath . Gen2.java*. To run the utility, the following command was used: *java -cp . Gen2 abso2*. This read in the file named *abso2* (the filename was taken as a parameter on the command line) and outputted a simpler file with only the pertinent information needed for simple case evaluation to a file called *inputb.out*. This file was then used by the genetic programming solution as the input on which to base tree building decisions.

This leaves all the final processing up to ECJ. ECJ becomes responsible for reading in the file, parsing the information into the appropriate variable arrays, and then iterating through the arrays in such a way that the information gets into the tree nodes at the proper time and proper place.

Because reading from a file can be an expensive operation, ECJ only opens the file once. It does this during an initial setup method and during this time stores all file values in the appropriate variable arrays. For instance the acknowledgment numbers are stored in a double array called *inputAck*. The state information on missing packets, essentially the answer key, is stored in a double array called *missing*.

All values used inside ECJ were of type double. This is to avoid errors in data caused by truncation. Because random ephemeral constants were of type double, the results of any of their computations should be capable of being passed back on up to their parent node which should also be of type double. Since during genetic programming nodes are mixed and matched, it was ideal to maintain commonality and uniformity among node value types. Therefore, having one node of type double is an indication that they should all be of type double.

Once all input values were represented in their proper variable arrays in memory, ECJ was ready to begin growing the trees.

Growing the Trees

The evolutionary concept decrees survival of the fittest, and that is what makes genetic programming so powerful. The basic concept underlying the methods used to evolve individuals is that nodes are randomly arranged into trees, and those arrangements which perform the best are saved and modified for successive generations. This ensures the propagation of stronger performers, closer fits, and eventually the winning ideal individual, which is basically the winning tree arrangement. A tree is an arrangement of nodes in such a way that the evaluation of the nodes is done in a specific order that means the tree can be translated into an equivalent mathematical expression where the internal nodes become the operators acting upon the terminal nodes which become the operands.

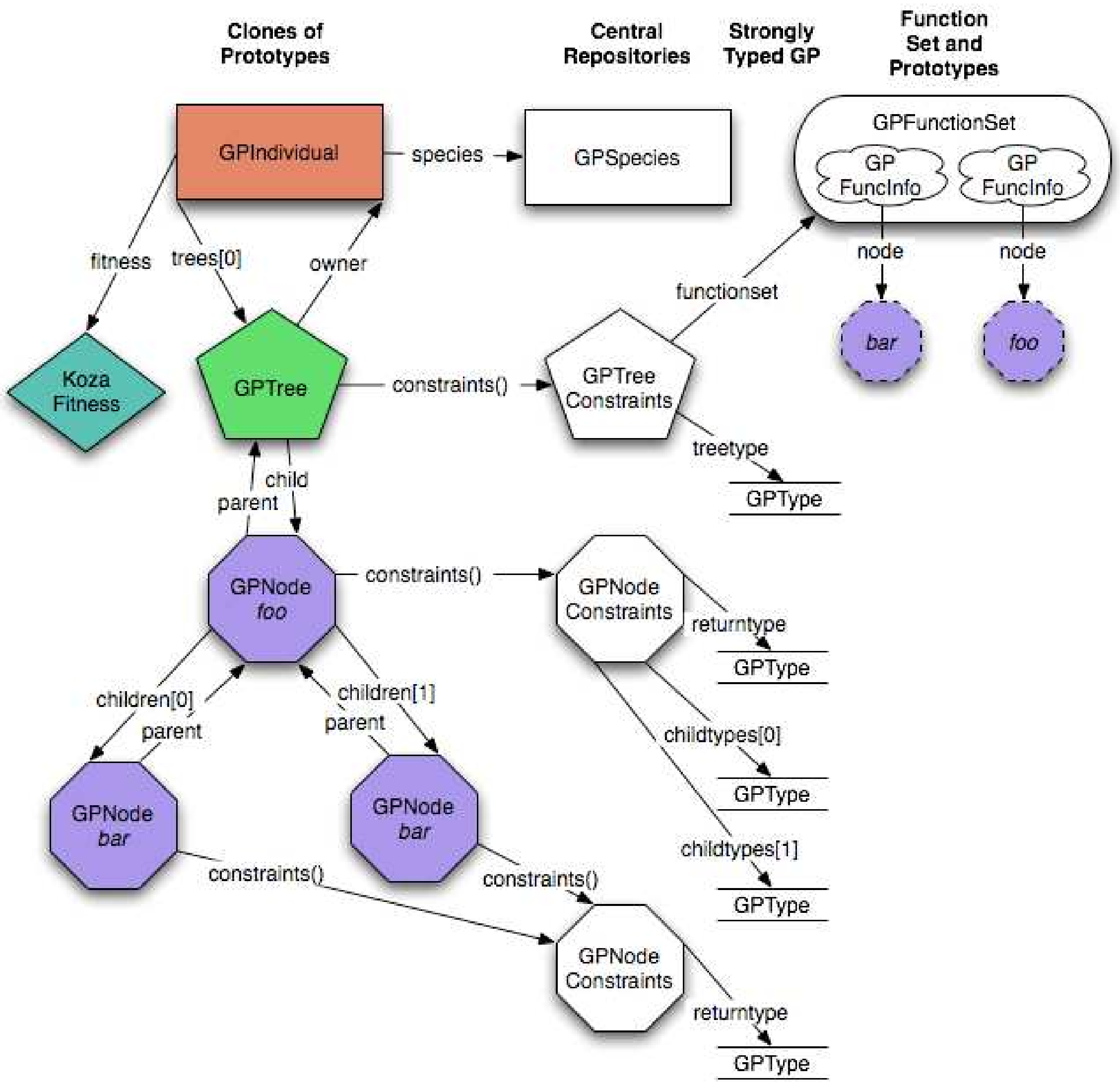


Figure 8: Representation of ECJ's genetic programming system
(ECJ:http://cs.gmu.edu/~eclab/projects/ecj/)

GPIndividual is a class that represents one individual (or one tree) within the total population which can consist of multiple trees. A tree is made up of an amalgam of nodes chosen from the function set and data object set, in essence, operators and variables. *GPNodeConstraints* defines the behavior of each node. The behavior of nodes belonging in the function set will differ from the behavior of data object nodes. The inner function set nodes, represented in the diagram above as the *foo* node, must evaluate both children before returning a value up the tree. The data object nodes, represented in the diagram above as *bar* nodes, simply return their value.

GPType is a field found in each node specifying the type of the return value. In the case of TCPMissing, *DoubleData* objects were used throughout the tree. *DoubleData* is a Java class that encapsulates values of the primitive data type *double*. To help avoid type mismatch, *GPTreeConstraints* also contains a *GPType* field. *GPType* guarantees that all nodes maintain uniformity within the tree by having each node return data of the same type. Every node added to the tree should return a value of one-and-the-same type, and in the case of TCPMissing, this type was DoubleData.

GPTreeConstraints also maintains a *GPFunctionSet* object. *GPFunctionSet* keeps track of all the nodes available for insertion into a tree. In the case of TCPMissing, these were the seven variable nodes and the three operator nodes. During the formation of trees, nodes were selected from this pool depending on whether an interior node was needed for operator functionality (*foo*) or whether a terminal node was needed to represent a variable (*bar*).

There are several important stages involved in evolving trees. First there is the generation of the initial population to consider. Then between successive generations, trees change according to mutations and crossovers and are selected for survival based on some fitness function. In ECJ, subclasses of *GPNodeBuilder* take care of the tree generation algorithms used in population initialization and mutation.

The initial population at the very beginning is generated in ECJ according to the principles of ramped half-and-half. There are two methods of growing a tree. The full method grows a full tree, meaning that each branch in the tree has a maximum possible depth. Given a maximum depth of D_{max}, all branches of the tree will be of size D_{max} with no variation in the size of each individual branch. The other method is the grow method where the tree size may vary from one node to as many nodes as will fit up to the limit of D_{max}. In ramped half-and-half, a population of trees is created where half the trees were grown using the full method and half the trees were grown using the grow method. Initially the population size was set to 1024 individuals. However, in order to widen the gene pool and see an increase in better results faster, the population size limit was increased by one order of magnitude to 10,240 individuals per generation.

Once a population exists, the individuals must be evaluated against a fitness function in case an ideal individual has evolved. Once an ideal individual is found, the problem is solved and ECJ’s work is done. If no ideal individual was found, ECJ goes through the process of creating the population that will comprise the next generation.

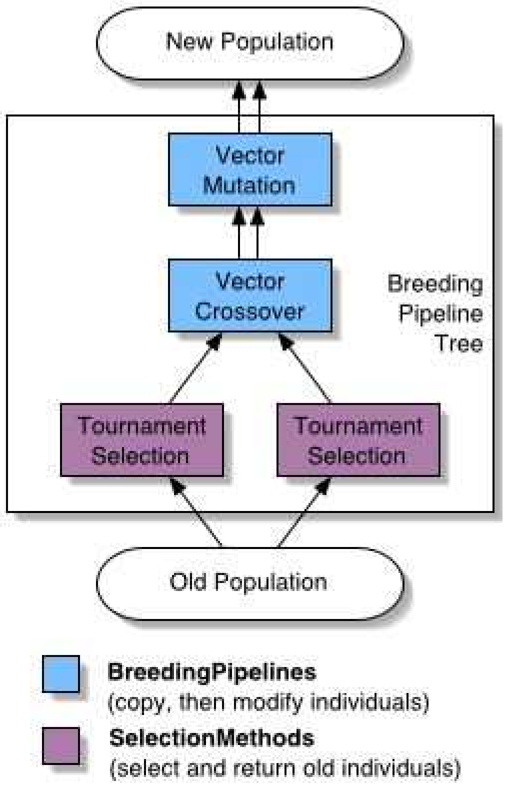


Figure 9: A breeding pipeline conceptualization of the methods used by ECJ to create a new population from an old one (ECJ:http://cs.gmu.edu/~eclab/projects/ecj)

Tournament selection is the selection method utilized by ECJ to pick the parents of the next generation. The variation of tournament selection used by ECJ picks seven possible parents from the total population, but this number can vary. The greater the tournament selection size, the greater the selection pressure. For *x* individuals picked at random, the larger the value of *x*, the greater the probability is that a good individual with a high fitness is among that group. This decreases the chance of a weak individual being chosen as a parent. A larger selection size will more quickly skew the selection of parents for recombination towards those with higher fitness values. A selection pressure that is too high is undesirable because it might lead towards convergence on a locally optimal solution and not the ideal globally optimal solution.

According to the rules of tournament selection, a number of random individuals are chosen from the population. These individuals are evaluated according to their fitness functions and the strongest (best-performing) individual is chosen as the winner.

The fitness function for TCPMissing was based on the number of correctly identified missing packets. A tree that is unable to identify the missing packets will have a bad fitness value. False positives, or identifying missing packets where none exist, can also adversely affect a tree's fitness. As part of evaluating the fitness for each tree, a training set of data is required where the correct answer for every corresponding line of input is predetermined. This is needed because the expected result must be available to the fitness function for its use in comparing the performance of each individual with the ideal performance. The training set of data for TCPMissing included both the input dataset and the expected results. This information was read into memory from a file created by *Gen2* at the start of the program.

From the figure above, it can be seen that tournament selection is performed twice during the breeding pipeline so that two parents can be chosen for recombination. The practiced method of recombination in ECJ is a subtree crossover. Picture a random node chosen from the tree and then envision the subtree represented with that node as its parent. If it is a leaf, there is only one node in the subtree, which is itself. Exchanging two randomly chosen subtrees of two individuals in one generation will create two new individuals for the population of the successive generation, hopefully creating trees with performance closer to the ideal individual.

Mutation is essentially creating a new tree from an old tree through some random small variation. A random node is chosen in the old tree and used as the point of mutation. First the subtree connected to that node is removed. Then a new randomly generated tree is grown to that point. There is a small caveat that goes along with this method of mutation. One needs to be aware that if left to themselves, trees tend to become successively larger throughout each generation. This is known as bloat, or survival of the fattest [Eibe1998]. In an effort to avoid this problem, a maximum tree size is defined according to maximum depth and all trees are prevented from growing beyond this.

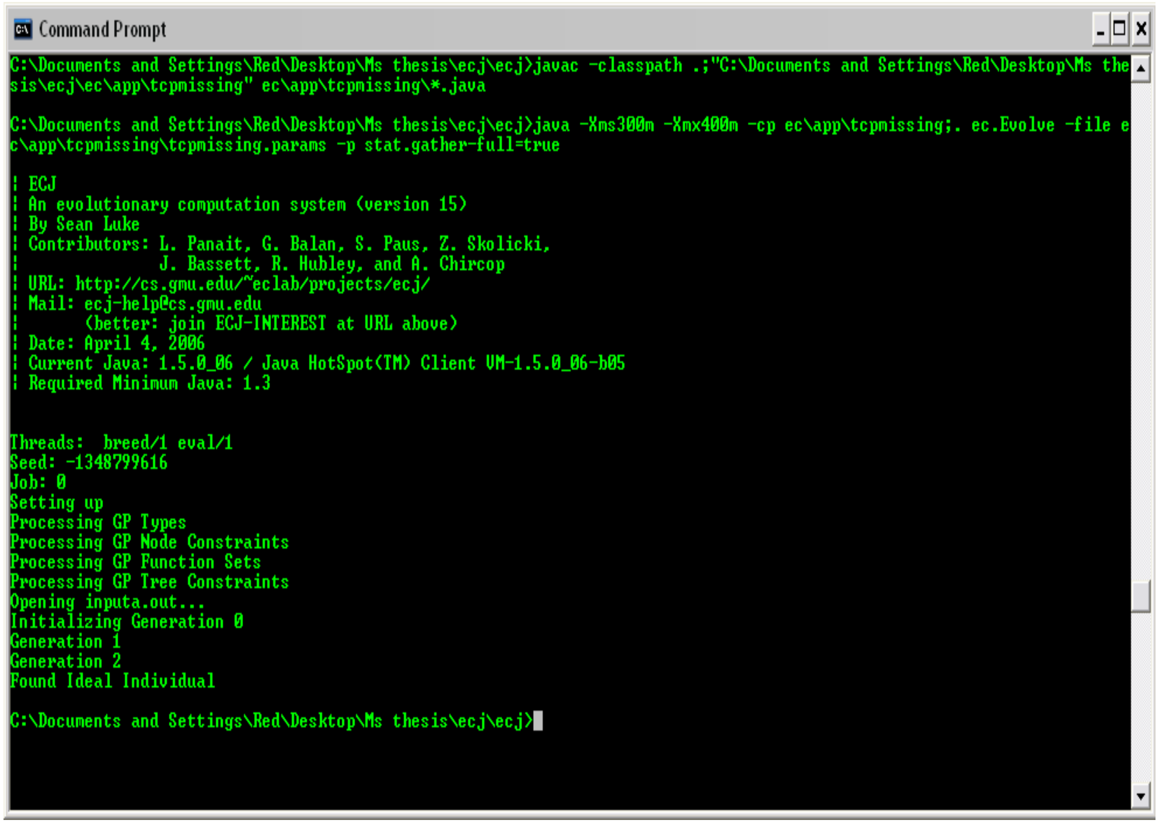
The selection and modification process represented by the breeding pipeline diagram can be repeated multiple times until sufficient individuals are created to populate the new generation. In TCPMissing, every generation consisted of 10,240 individuals. When the number of individuals in the new generation reaches the population size limit, the breeding stage completes, and the new generation is evaluated to check for the existence of an ideal individual. If none is found, then the current generation becomes the old generation and a new generation must be created. This cycle continues until either an ideal individual is found or the maximum number of generations is reached. In TCPMissing, the maximum number of generations was set to fifty. However, processing rarely continued on for that long as ideal individuals were usually found within the first handful of generations.

Execution of ECJ on the TCPMissing Problem

The following is the output of the session where an ideal individual was found. First the program is compiled and then executed. After two generations, an ideal individual was found. To avoid large amounts of data from scrolling across the screen, ECJ outputs data to a file named *out.stat*, which can be found in Appendix B.

>>javac -classpath .;[location of ecj] ec\app\tcpmissing\*.java

>>java -cp ec\app\tcpmissing;. ec.Evolve -file ec\app\tcpmissing\tcpmissing.params -p stat.gather-full=true



```
C:\Documents and Settings\Red\Desktop\Ms thesis\ecj\ecj>javac -classpath .;"C:\Documents and Settings\Red\Desktop\Ms the
sis\ecj\ec\app\tcpmissing" ec\app\tcpmissing\*.java

C:\Documents and Settings\Red\Desktop\Ms thesis\ecj\ecj>java -Xms300m -Xmx400m -cp ec\app\tcpmissing;. ec.Evolve -file e
c\app\tcpmissing\tcpmissing.params -p stat.gather-full=true

| ECJ
| An evolutionary computation system (version 15)
| By Sean Luke
| Contributors: L. Panait, G. Balan, S. Paus, Z. Skolicki,
|               J. Bassett, R. Hubley, and A. Chircop
| URL: http://cs.gmu.edu/~eclab/projects/ecj/
| Mail: ecj-help@cs.gmu.edu
|         (better: join ECJ-INTEREST at URL above)
| Date: April 4, 2006
| Current Java: 1.5.0_06 / Java HotSpot(TM) Client VM-1.5.0_06-b05
| Required Minimum Java: 1.3


Threads:  breed/1 eval/1
Seed: -1348799616
Job: 0
Setting up
Processing GP Types
Processing GP Node Constraints
Processing GP Function Sets
Processing GP Tree Constraints
Opening inputa.out...
Initializing Generation 0
Generation 1
Generation 2
Found Ideal Individual

C:\Documents and Settings\Red\Desktop\Ms thesis\ecj\ecj>
```

Figure 10: Screenshot of ECJ running on the TCPMissing problem

Once an ideal individual representing a solution equation is found, a small Java program was built around this equation and consisted of three main components. This Java program is contained in a file called *GPSolution.java* that is included in Appendix B. The first component of this class was responsible for reading in the dataset (in this case the trace file of network traffic). The second component did the heavy lifting of using the inputs as values to evaluate the equation, multiple times if necessary. For instance, an input file of size *n* packets would require *n* evaluations of the genetic programming equation. This means that a genetic programming solution for the TCPMissing problem would have a complexity of O(n). The following was an equation discovered by ECJ to solve for missing packets:

$$\begin{gathered}
(((((\mathit{Seq} - \mathit{M0Length}) - (\mathit{M0Seq} - 0.5915718)) * \\
((\mathit{Seq} - -0.11322764) - (\mathit{M0Ack} * \mathit{M0Seq}))) * (((\mathit{Seq} * \mathit{Seq}) \\
- (-0.075646505 + ((\mathit{M0Ack} * \mathit{M0Length}) - (\mathit{M0Ack} * \mathit{Seq})))) * \\
((0.56651837 - \mathit{Ack}) * (\mathit{Ack} + \mathit{M0Length}))))))
\end{gathered}$$

The third component was responsible for displaying the appropriate output and results. This implementation of the genetic programming solution allowed testing of the equation on other datasets to determine the generality (and correctness) of the solution.

It was possible to start over from the original input file and remove a different number of packets from different locations and still receive the correct answer. This means that not only was the GP equation correct, but it was also general enough to allow for different packets than the ones it was specifically trained on to go missing and still pick up on that fact.

Results

The methodology for determining the accuracy of the system was much the same as it was for the non-learning component. It originally used trace files where the number of missing packets was pre-identified, removed according to pre-determined percentages. The percentages of missing packets removed and the percentages of missing packets identified by the genetic programming solution correspond exactly, even better than the results of the non-learning component which were approximate. The genetic programming solution was also able to perform better with a higher percentage of missing packets. This makes sense if it was able to extrapolate the underlying working mathematical model. As if these benefits were not enough, the genetic programming solution was also easier and simpler to implement from a programmer's perspective.

To achieve good results, it helped greatly to increase the generations' population size. This value was set to 10,240, an order of magnitude increase from the default value. This increase in population size greatly increased the efficiency of the trees. This makes sense because there were many more individuals to choose from for reproduction and mutation.

The randomness of the system was also different for each run. ECJ utilizes a seed parameter which sets up the starting point for the MersenneTwisterFast algorithm. This random number generator has an extremely long period, which means that patterns are less likely to be discernable. Setting the seed to be the current time at the moment of execution means that every run on the same problem will generate different random trees. Since ECJ is fully deterministic, specifying a specific numeral value for the seed will generate the same trees across different runs.

This work contributes greatly to the field of intrusion detection because it develops an intrusion detection component which is able to act on stored TCP trace files to determine after-the fact whether packets which should be in the stream are missing, or whether they were dropped by the network. This can be very important for post-attack analysis.

The genetic programming model is useful because it demonstrates proof of concept that a solution can be found for determining missing packets from the simplest of input streams. The attractive feature of genetic programming is since the solutions are based on mathematical models, they have the potential to achieve higher rates of accuracy than systems based on sequential algorithms if all the special cases are not selectively pre-programmed.

One of the nice things about using ECJ as the gateway to a genetic programming solution is code reuse. The core functionalities of genetic programming are already coded and modularized. To make use of these functionalities requires minimum changes to the program. To increase the node count, additional node classes were written. However, beyond that only two files needed to be modified: *MultiValuedRegression.java* and *tcpmissing.params*, and both of these files were included in Appendix B. Essentially all specifics of the particular implementation are coded in these two files. And in this case, instead of writing code to solve the actual problem, one must simply write code to describe the problem. This difference is important, because especially for difficult problems, it is much simpler to describe the problem than to write the code to exhaustively solve it. And indeed, the genetic programming solution was able to find an ideal individual (a regression across the data points) within about two seconds or two successive generations. This makes genetic programming faster and cheaper.

It is also more efficient. Whereas a programmer must explicitly insert code for each possible scenario, genetic programming handles the subtle cases automatically. However, one thing to be aware of in evolving a tree is the completeness of training data. If the training dataset does not represent all possible aspects of the problem, the genetic programming solution might not recognize those missing aspects if they are introduced later. In addition, the training dataset must not contain erroneous data that could crop up due to faulty preprocessing modules. One way to test input files to ECJ is to run similar files through the TCPMissing algorithm, and see if results are similar.

Another thing to watch out for in regards to the training dataset is that there are no accidental patterns set up that could distract the genetic programming solution away from the real problem. One problem experienced early on was that the ideal individual found during genetic programming could not perform correctly on different input files. If it was presented with data other than the data it was trained on, it could not generalize due to the fact that it had focused on a pattern not necessary to the problem, such as exactly every third packet was removed. That is why it is important to make sure the training dataset is complete and fair as a truly random and representative subsection of the problem space.

This presents one area of possible future work: that of moving beyond proof-of-concept to a full-bodied implementation of a genetic programming solution that was generalized enough to handle all possible special case scenarios. This could be done either by training on a completely generalized dataset, which might be a difficult determination to make. Or the special cases could be slowly introduced. Using the simplest case as the starting point, test the solution equation on a file representing a special case not yet introduced to the system. If the missing packets are correctly identified, good! Otherwise use the mistakes as a starting point for successive genetic programming solutions for those special case scenarios.

Another area of future work consists of extending the functionality of the GP solution to be able to account for both the source and destination machines involved in a single flow, or even extending it to account for multiple flows. However, this functionality is easy to implement using a preprocessing module (the method adopted by the GP solution) so it might not offer a great enough pay-off to make it worthwhile to pursue.

CHAPTER 4

A COMPARISON OF BOTH APPROACHES

TCPMissing is an IDS analytical component used to evaluate trace files in post-attack forensics to determine the number of missing packets. Knowing packets are missing from the traffic log files can help investigators avoid mistakes made based on incomplete information.

This thesis has taken an in depth look at the development of an IDS component for the determination of missing packets according to two different approaches. The first algorithmic approach stemmed from traditional computer science techniques and attempted to determine missing packets by identifying special case scenarios and applying separate formulas to each case. The second approach simply ran the data through a mathematical equation determined according to the rules and results of genetic programming.

It was through the influence of Denning, Bace, and Alder that a genetic programming solution was developed for TCPMissing. Due to the complex nature of TCP, explicitly coding an algorithmic solution for every single special case scenario turned out to be a daunting task. This challenge prompted a switch in direction as encouraged by Bace with her advice to remain flexible and open to new possibly better routes. Drawing upon the success of Denning with her expert system IDS, an AI solution was sought which would be faster, cheaper, and more efficient than the existing algorithmic method. Alder's advice on presenting research results in an understandable, palatable form helped guide the writing of this thesis.

Ease of Implementation

Genetic programming presented the AI alternative to the algorithmic approach. Using ECJ as the workhorse to perform most of the genetic programming functionalities (such as representation, selection, and mutation), the TCPMissing problem was encoded. It was easier and cheaper in man-hours to encode the *problem* of TCPMissing (as used in the GP solution) than to encode an actual algorithmic conclusive *solution* to the problem of TCPMissing.

Encoding the problem in ECJ involved specifying the inputs and the expected result that corresponded to that set of inputs. In the case of TCPMissing, this input file consisted of a number of packets and the corresponding expected result was a number on each line indicating whether a packet had been removed or not from the previous line. Then the rest was up to ECJ to correctly determine the proper regression on the dataset.

Although a GP implementation might prove a bit tricky to a first time user of ECJ, it was still much simpler than the algorithmic implementation of TCPMissing. In order to successfully implement a solution to TCPMissing using the algorithmic approach, one must separately encode every possible special case scenario that might occur under the umbrella of complex behaviors of TCP. In essence, one must program the complete solution instead of simply defining the complete problem as in genetic programming.

Accuracy

The GP solution was able to demonstrate more accurate results than the algorithmic approach. Both implementations were tested on the file “*real1b*” by successively removing different packets from the same file. The GP solution was able to keep up with differences in the location of the removed packets better than the algorithmic approach. This is because the genetic programming approach keeps creating successive generations of individuals until an ideal individual is found. Ideal individuals are those which are able to correctly identify all missing packets with no false positives. This means that they perform perfectly on the training dataset. It is part of creating a good dataset which provides generality to the solution so that it can also perform well on a different dataset.

This was tested on the GP solution. The resulting equation represented by the ideal individual was tested on different datasets than the one specifically used to train it. It performed as expected with perfect accuracy in predicting missing packets.

The GP solution was also more efficient. The complexity of evaluating a mathematical expression can be modeled as O(n) where n is the number of packets. An equivalent execution of the algorithmic solution would require the traversal of several possible branches of thought based upon the relevant special case scenarios at play. If m tasks or comparisons must be performed upon a packet to cater to the spectrum of possible special case scenarios, then the complexity for the algorithmic approach becomes $O(m \times n)$, a worse complexity than that of the GP solution.

Conclusion

In conclusion, genetic programming held the solution to a difficult problem involving the algorithmic implementation of TCPMissing. The problem occurred because of the plethora of special case scenarios that arise out of the complexities of TCP. Where genetic programming won out over the algorithmic implementation was that genetic programming requires the encoding of just the problem, as opposed to the encoding of the entire solution. The genetic programming solution was shown to be easier to implement, more accurate, and more efficient.

REFERENCES

[Alde2006] Alder, Raven. Telephone Interview with Rebekah Black. 1 Mar 2006.

[Ande1980] Anderson, James. “Computer Security Threat Monitoring and Surveillance.” Technical report, James P Anderson Co., Fort Washington, Pennsylvania. 26 February 1980.

[Axel2000] Axelsson, Stefan. “Intrusion Detection Systems: A Survey and Taxonomy.” Goeteborg, Chalmers University of Technology, Department of Computer Engineering. 14 March 2000.

[Bace2006] Bace, Rebecca. Telephone Interview with Rebekah Black. 10 March 2006. 9:30-10:00am.

[Brow2000] Brown, John Seely, and Paul Duguid. "Mysteries of the Region: Knowledge Dynamics in Silicon Valley." The Silicon Valley Edge. Stanford University Press p. 16-39 (2000). 6 Feb 2006 <http://www.sociallifeofinformation.com/Mysteries_of_the_Region.htm>.

[Brow2000] Brown, John Seely, and Paul Duguid. "Ideas to Feed Your Business: Re-Engineering the Future." The Internet Standard 24 April 2000. 6 Feb 2006 <http://www.thestandard.com/article/0,1901,14013,00.html>.

[Cann1998] Cannady, James. "Artificial Neural Networks for Misuse Detection**."** National Information Systems Security Conference, NISSC. October 5-8 1998. Arlington, VA. p. 443-456.

[Cann2000] Cannady, James. "Next Generation Intrusion Detection: Autonomous Reinforcement Learning of Network Attacks." 23rd National Information Systems Security Conference, NISSC. 2000. <http://csrc.nist.gov/nissc/2000/proceedings/papers/033.pdf. >.

[Cars2003] Carstens, Tim. "Programming with Pcap." Spring 2003. <http://www.tcpdump.org/pcap.htm>, <http://JPCAP.sourceforge.net/>.

[Dass2003] Dass, Mayukh. "LIDS: A Learning Intrusion Detection System." MS. Thesis, University of Georgia, 2003.

[Davi2005] Davies, Jason. "Optimizing Java for Speed." Netspade. Copyright 2005. October 2005. <http://www.netspade.com/articles/java/optimizing/speed.xml>.

[Deba1992] Debar, H., Becke, M., & Siboni, D. "A Neural Network Component for an Intrusion Detection System." IEEE Computer Society Symposium on Research in Security and Privacy. 1992. p 240-250.

[Denn1987] Denning, Dorothy E. "An Intrusion Detection Model." IEEE Transactions on Software Engineering, Vol. SE-13, No. 2, p. 222-232, Feb 1987.

[Denn1996] Denning, Dorothy E. "The Future of Cryptography." Georgetown University. 6 Jan 1996. 1 Feb 2006 <http://www.cosc.georgetown.edu/~denning/crypto/Future.html>.

[Denn2006] Denning, Dorothy. Telephone Interview with Rebekah Black. 27 Jan 2006.

[Eibe2003] Eiben, A. E., and J. E. Smith. “Introduction to Evolutionary Computing.” Verlag Berlin Heidelberg: Springer, 2003. p 101-113.

[Ever2005] Evers, Joris. "Hackers rally behind Cisco flaw finder." ZDNet 1 Aug 2005. 06 Mar 2006 <http://www.zdnet.com.au/news/security/soa/Hackers_rally_behind_Cisco_flaw_finder/0,2000061744,39205047,00.htm>.

[ECP2006] "Technology Jobs for Women: Women and Their Role in the Development of the Modern Computer: Computer Wonder Women." Educational CyberPlayground (ECP): founded by Karen Ellis. 08 Feb. 2006 <http://www.edu-cyberpg.com/pdf/cwomen.pdf>.

[Fran1994] Frank, Jeremy. “Artificial Intelligence and Intrusion Detection: Current and Future Directions.” Proceedings of the 17th National Computer Security Conference. Baltimore, MD, pp 22-33, October 1994.

[Fuji2006] Fujii, Keita. JPCAP ver.0.5. Copyright (c) 2006. 09 April 2006. <kfujii@ics.uci.edu> <http://netresearch.ics.uci.edu/kfujii/JPCAP/doc/javadoc/index.html>.

[Gosl1996] Gosling, James, and Henry McGilton. "The Java Language Environment - A White Paper." Sun Developer Network. May 1996. Sun Microsystems. Fall 2005 <http://www.javasoft.com/docs/white/langenv>.

[Gray2004] Patrick Gray and Fran Foo. "Hackers: Under the Hood." ZDNet Australia 19 Apr 2004. 06 Mar 2006 <http://www.citationmachine.net/index.php?mode=form&g=6&list=nonprint&cm=12>.

[Ilgu1992] Ilgun, Korel. USTAT - A Real-time Intrusion Detection System for UNIX. Master's Thesis, University of California at Santa Barbara, November 1992.

[Katz2001] T. Katzlberger, G. Biswas, J. Bransford, and D. Schwartz, and TAG-V, “Extending Intelligent Learning Environments with Teachable Agents to Enhance Learning,” Tenth Intl. Conf. on AI in Education: AI-ED in the Wired and Wireless Future, J.D. Moore, C.L. Red.eld, and W.L. Johnson, eds., IOS Press, Amsterdam, p. 389-397, May 2001.

[Kear2005] Kearns, Dave. "The Man-in-the-Middle gets Caught up in ID Theft." Network World 26 October 2005. 26 Feb 2006 <http://www.networkworld.com/newsletters/dir/2005/1024id2.html>

[Kuma1995] Kumar, Sandeep. “Classification and Detection of Computer Intrusions.” Ph.D. Diss. Department of Computer Sciences, Purdue University. August 1995.

[Mosk2002] Moskal, Barbara M. "A Summary of Results from the Survey of the Earned Doctorate: Women Earning Computer Science Doctorates." Computing Research News, p. 2, 11. May 2002. 26 Mar 2006 <http://www.cra.org/CRN/articles/may02/moskal.html>.

[Penn2003] Pennington, Adam G. Strunk, John D. Griffin, John Linwood. Soules, Craig A.N. Goodson, Garth R. Ganger, Gregory R. "Storage-based Intrusion Detection: Watching Storage Activity for Suspicious Behavior." Carnegie Mellon University. Proceedings of the 12th USENIX Security Symposium, Washington, DC. August 2003.

[Puke1996] Puketza, N. Zhang, K. Chung, M. Mukherjee, B. Olsson, R. A. "A methodology for Testing Intrusion Detection Systems." IEEE Transactions on Software Engineering, 22(10), pp. 719-729, Oct. 1996.

[RFC71981] "RFC 793: Transmission Control Protocol." DARPA Internet Program Protocol Specification. September 1981. <http://www.ietf.org/rfc/rfc0793.txt>.

[Rowe2006] Rowe, Neil C., and Sandra Schiavo. "An Intelligent Tutor for Intrusion Detection on Computer Systems." Computers and Education. 1998. 6 Feb 2006 <http://www.cs.nps.navy.mil/people/faculty/rowe/idtutor.html>.

[Shen2003] Sheng, Lu. Jian, Gong. Suying,Rui. "A Load Balancing Algorithm for High Speed Intrusion Detection." Nanjing, China, Southest University, Department of Computer Science and Engineering. Eastern China (North) Network Center of CERNET. 2003.

[Slas2004] Human Interest Dept., posted by timothy. "Hackers: Under the Hood." Online posting. 20 Apr 2004. Slashdot. 06 Mar 2006. <http://developers.slashdot.org/article.pl?sid=04/04/19/2353230&tid=>.

[Snif2004] Sniffen, Michael. "Privacy Protecting Programs Killed." Associated Press, CastleCops, 15 Mar 2004. 1 Feb 2006. <http://castlecops.com/modules.php?name=News&file=article&sid=4958>

[Thie2002] Thieme, Richard. "The IDS Den Mother." Information Security April 2002. 26 Mar 2006 <http://infosecuritymag.techtarget.com/2002/apr/qa.shtml>.

[vanM2000] van Millingen, Liz. "Confidence to Control – Attitudes Towards Technology Based on a Developing Self-Identity." British Psychological Society, London Conference, poster presentation, Dec 2000. 08 Apr 2006 <http://www.le.ac.uk/psychology/eavm1/poster.london2000.html>.

APPENDIX A

TCPMISSING API DOCUMENTATION

Tcpmissing

Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class Tcpmissing

```
java.lang.Object
  |
  +--Tcpmissing
```

All Implemented Interfaces:

JPCAPHandler

class **Tcpmissing**
extends Object
implements JPCAPHandler

Program Name: TCP Missing

Author: Rebekah Black

Date: Spring 2004

Purpose: To determine the percentage of missing packets due to application failure as opposed to network loss.

Class: Tcpmissing (where it all begins and eventually ends, like every good thing...)

Field Summary

`(package private)`	**`ARPs`** number of ARP packets

static long	
(package private) static int	**delayed** number of delayed packets
(package private) static int	**dropped** number of dropped packets
(package private) static int	**duplicate** number of duplicate packets
(package private) static int	**flows** number of flows in trace file
(package private) static long	**ICMPs** number of ICMP packets
(package private) static int	**luckyPacket** determines which packet gets removed
(package private) static Hashtable	**map** stores the seperate unprocessed flows
(package private) static int	**miss** how many packets get removed in any given run-through
(package private) static int	**missing** number of missing packets
(package private) static long	**other** number of other currently non-processable type packets
(package private) static long	**TCPs** number of TCP packets
(package private) static long	**total** total number of processed packets
(package private) static long	**UDPs** number of UDP packets
(package private) static int	**whosTurn** tracker of the lucky packet

Constructor Summary

(package private)	**Tcpmissing**()

Method Summary

void	**addToFlows**(IPPacket packet) Adds the TCPPacket to the appropriate flow
static void	**checkParams**(String[] args) Checks the incoming command line parameters from main
boolean	**filter**() Filter option to facilitate manual removing of packets
static void	**finish**() Processes any unfinished flows once all packets have been recieved
void	**handlePacket**(Packet packet) A JPCAP instance must have first been created to generate the packets passed in
static void	**main**(String[] args) Opens a file, parses packets, and returns results
static void	**printStats**() Preferably called at the end of the program when variable values have finished changing
boolean	**saveResults**(String name) Callable from main, brings in third command line parameter as the first part of the new file name

Methods inherited from class java.lang.Object
, clone, equals, finalize, getClass, hashCode, notify, notifyAll, registerNatives, toString, wait, wait, wait

Field Detail

total

`static long` **`total`**

total number of processed packets

ARPs

`static long` **`ARPs`**

number of ARP packets

TCPs

`static long` **`TCPs`**

number of TCP packets

UDPs

`static long` **`UDPs`**

number of UDP packets

ICMPs

`static long` **`ICMPs`**

number of ICMP packets

---

other

`static long` **`other`**

number of other currently non-processable type packets

---

missing

`static int` **`missing`**

number of missing packets

---

dropped

`static int` **`dropped`**

number of dropped packets

---

duplicate

`static int` **`duplicate`**

number of duplicate packets

---

delayed

`static int` **`delayed`**

number of delayed packets

---

flows

`static int` **`flows`**

number of flows in trace file

---

luckyPacket

`static int` **`luckyPacket`**

determines which packet gets removed

---

whosTurn

`static int` **`whosTurn`**

tracker of the lucky packet

---

miss

`static int` **`miss`**

how many packets get removed in any given run-through

---

map

`static Hashtable` **`map`**

stores the seperate unprocessed flows

Constructor Detail

Tcpmissing

`Tcpmissing``()`

Method Detail

handlePacket

`public void` **`handlePacket`**`(Packet packet)`

A JPCAP instance must have first been created to generate the packets passed in

Specified by:

`handlePacket` in interface `JPCAPHandler`

Parameters:

`Packet` - packet

Returns:

classifies the packet according to type. If it is a TCP packet, it is sent on in the program. Otherwise it dies.

---

main

```
public static void main(String[] args)
                 throws IOException
```

Opens a file, parses packets, and returns results

Parameters:

`args[0]` - trace data file name
`args[1]` - keyword "where" : which packet to eliminate, optional with 3 and 4
`args[2]` - int value of which packet to get rid of
`args[3]` - keyword "num" : how many packets to remove, optional with 1 and 2
`args[4]` - int value of how many packets to remove
`args[5]` - keyword "stats" : optional

Returns:
parses through the data file, filtering when necessary, and computes then displays the percentage of missing packets.

---

addToFlows

`public void **addToFlows**(IPPacket packet)`

Adds the TCPPacket to the appropriate flow
Parameters:
`IPPacket` - must be of type TCPPacket
Returns:
recieves new packets into the correct hashmap flow. If the flow has been completed and processed, it is removed from the hashmap

---

finish

`public static void **finish**()`

Processes any unfinished flows once all packets have been recieved
Returns:
processes any flows left in the hashmap

---

checkParams

`public static void **checkParams**(String[] args)`

Checks the incoming command line parameters from main
Parameters:
`args[0]` - trace data file name
`args[1]` - keyword "where" : which packet to eliminate, optional with 3 and 4
`args[2]` - int value of which packet to get rid of
`args[3]` - keyword "num" : how many packets to remove, optional with 1 and 2
`args[4]` - int value of how many packets to remove
`args[5]` - keyword "stats" : optional
Returns:
sets variables accordingly

---

saveResults

`public boolean **saveResults**(String name)`

Callable from main, brings in third command line parameter as the first part of the new file name
Parameters:
`name` - becomes part of writing file name and signifies the number of packets to remove
Returns:
opens a file to append the necessary information to the end, returns true if successful

---

filter

`public boolean **filter**()`

Filter option to facilitate manual removing of packets
Returns:
evenly removes 'miss' number of packets starting from 'luckyPacket' going in increments of 4000/miss, returns true if successful

---

printStats

`public static void **printStats**()`

Preferably called at the end of the program when variable values have finished changing
Returns:
prints out the total number of different packet types

Flow

Class Flow

```
java.lang.Object
  |
  +--Flow
```

class **Flow**
extends Object

Program Name: TCP Missing

Author: Rebekah Black

Date: Spring 2004

Purpose: To determine the percentage of missing packets due to application failure as opposed to network loss.

Class: Flow (logical representation of a flow and methods to analyze it for missing packets), depends on Tcpmissing as the driving end

Field Summary

private long	**dest** unique hash of destination machine and port
int	**fin** number of recieved fins in this flow
private IPPacket[]	**lastPacket** keeps track of the last recieved packets for each side
int	**nextSpot** pointer to the next place in the array of past packets
private IPPacket	**p** keeps track of current working packet
Vector	**packets** number of packets in this flow
private	**PredictedAck**

long[]	keeps track of each side's expected ack num
private int	**previous** keeps track of the index of the previous packet from same source
private long	**source** unique hash of source machine and port
private int	**them** identifier for the other side
static int	**totalFlows** tracks the total number of flows processed during one execution time
private int	**us** identifier for this connection
private Iterator	**v** iterates through all the packets in this flow

Constructor Summary

Flow() Initializes the variables needed to represent a flow

Method Summary

String	**addPacket**(IPPacket packet) Adds the packet to the Vector packets
void	**analyze**() (Packet) p must point towards a packet, should call getNextPacket() first
void	**finCase**() Handles the fin case
int	**findPrevious**() Called at least after the first packet has been inserted so that nextSpot does NOT equal 0, bad...
boolean	**getNextPacket**() A wonderfully robust function that takes care of most bookkeeping matters that change with each new packet, retrieves the next packet from the packets Vector
void	**norm**() Handles most intermediate cases, sets the predictedack for the other guy and compares its own predictedack.

boolean	outOfOrder() Handles out of order packets
void	print() Prints packets for debug purposes
void	processFlow() Begin analyzing this flow, one packet at a time
void	synCase() Handles the syn case

Methods inherited from class java.lang.Object
`, clone, equals, finalize, getClass, hashCode, notify, notifyAll, registerNatives, toString, wait, wait, wait`

Field Detail

fin

`public int` **`fin`**

number of recieved fins in this flow

---

packets

`public Vector` **`packets`**

number of packets in this flow

---

totalFlows

`public static int` **`totalFlows`**

tracks the total number of flows processed during one execution time

---

PredictedAck

`private long[]` **`PredictedAck`**

keeps track of each side's expected ack num

---

them

`private int` **`them`**

identifier for the other side

---

us

`private int` **`us`**

identifier for this connection

---

v

`private Iterator **v**`

iterates through all the packets in this flow

---

p

`private IPPacket p`

keeps track of current working packet

---

lastPacket

`private IPPacket[] lastPacket`

keeps track of the last recieved packets for each side

---

nextSpot

`public int nextSpot`

pointer to the next place in the array of past packets

---

source

`private long source`

unique hash of source machine and port

---

dest

`private long dest`

unique hash of destination machine and port

---

previous

`private int previous`

keeps track of the index of the previous packet from same source

Constructor Detail

Flow

`public Flow()`

Initializes the variables needed to represent a flow

Method Detail

addPacket

`public String addPacket(IPPacket packet)`

Adds the packet to the Vector packets

Parameters:

`packet` - must be of type TCPPacket

Returns:

puts the packet into the flow and analyzes it if the packet is a reset packet or second fin, returns "done" if the flow has been processed, otherwise returns "not yet"

---

processFlow

`public void processFlow()`

Begin analyzing this flow, one packet at a time

Returns:

increments number of flows in Tcpmissing, analyzes any packets available for analyzation

---

getNextPacket

`public boolean getNextPacket()`

A wonderfully robust function that takes care of most bookkeeping matters that change with each new packet, retrieves the next packet from the packets Vector

Returns:

returns true if another packet was retrieved successfully, false otherwise

---

findPrevious

`public int findPrevious()`

Called at least after the first packet has been inserted so that nextSpot does NOT equal 0, bad...

Returns:

returns the index in the lastPacket array where the previous packet from the same source is stored, -1 otherwise

---

analyze

`public void analyze()`

(Packet) p must point towards a packet, should call getNextPacket() first

Returns:

processes the packet according to whether it is a special condition (syn, fin) or normal (norm)

---

synCase

`public void synCase()`

Handles the syn case

Returns:

sets the predictedack for the other guy as its seq# + 1

---

finCase

`public void finCase()`

Handles the fin case

Returns:

sets the predictedack for the other guy as its seq# + 1 if the length of the packet is zero, otherwise it is set to seq# + length

---

norm

`public void **norm**()`

Handles most intermediate cases, sets the predictedack for the other guy and compares its own predictedack. If a discrepancy is found and no explaining special case exists for it, the missing counter is incremented

---

outOfOrder

`public boolean **outOfOrder**()`

Handles out of order packets

Returns:

a special case which returns true if a packet was recieved out of order

---

print

`public void **print**()`

Prints packets for debug purposes

APPENDIX B

SELECT CODE FROM THE GP IMPLEMENTATION OF TCPMISSING

GPSolution.java

```
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;
import java.util.StringTokenizer;
import java.util.Stack;
import java.io.FileReader;

/**
 * ---------------------------------------------------------------------------
 * This program takes the input of a traffic capture file and outputs
 *  the number of missing packets.
  *
 * @version 1.0
 * @author  Rebekah Black
 * ---------------------------------------------------------------------------
 */

public class GPSolution {

    public static final int SIZE_DATA = 36; // the number of packets read in from file

    public static double[] missing= new double[SIZE_DATA];
    public static double currentDeltaT=0;
    public static double[] inputDeltaT= new double[SIZE_DATA];
    public static double currentID=0;
    public static double[] inputID= new double[SIZE_DATA];
    public static double Length=0;
    public static double[] inputLength= new double[SIZE_DATA];
    public static double currentFlag=0;
    public static double[] inputFlag= new double[SIZE_DATA];
    public static double currentSrcDst=0;
    public static double[] inputSrcDst= new double[SIZE_DATA];
    public static double Seq=0;
    public static double[] inputSeq= new double[SIZE_DATA];
    public static double Ack=0;
    public static double[] inputAck= new double[SIZE_DATA];
    public static double M0Length=0;
    public static double M0Seq=0;
    public static double M0Ack=0;


    public static void setup(String filename)
       {

    try{
    //This is where I'm going to open the input file and read it in.
       System.out.println("Opening ..."+filename);

       FileReader inputFileReader   = new FileReader("D:\\thesis\\data\\"+filename);

    // Create Buffered/PrintWriter Objects
       BufferedReader inputStream   = new BufferedReader(inputFileReader);

       String inLine = null;
       String temp = null;
       StringTokenizer st = new StringTokenizer(temp=new String());
       int count = 0;

    while ((inLine = inputStream.readLine()) != null && count < SIZE_DATA) {
         st = new StringTokenizer(inLine, " \t");
         missing[count]=(new Double(st.nextToken())).doubleValue();
         //System.out.print("missing: "+missing[count]);
         inputDeltaT[count] = (new Double(st.nextToken())).doubleValue();
         inputID[count] = (new Double(st.nextToken())).doubleValue();
         inputLength[count] = (new Double(st.nextToken())).doubleValue();
         inputSrcDst[count] = (new Double(st.nextToken())).doubleValue();
         temp = st.nextToken();
         if(temp.equalsIgnoreCase("S"))
           inputFlag[count]=0;
         else if(temp.equalsIgnoreCase("F"))
           inputFlag[count]=1;
         else if(temp.equalsIgnoreCase("R"))
           inputFlag[count]=2;
         else //if(temp.equalsIgnoreCase("P"))
           inputFlag[count]=3;
       //else inputFlag[count]=-1;
       inputSeq[count] = (new Double(st.nextToken())).doubleValue();
       inputAck[count] = (new Double(st.nextToken())).doubleValue();
       //System.out.print(" ack: "+inputAck[count]);
       count++;
       //System.out.println("count: "+count);

    }//while
      } catch (IOException e) {

        System.out.println("IOException:");
        e.printStackTrace();
      }
    }

    public static void evaluate()
      {

        int hits = 0;
        double sum = 0.0;
        double expectedResult=0;
        double calcMiss=0;

       for(int i =0; i < SIZE_DATA; i++){
            currentDeltaT = inputDeltaT[i];
            currentID = inputID[i];
            currentFlag = inputFlag[i];

            if(currentSrcDst == 0){
              M0Seq  = Seq;
              M0Ack  = Ack;
              M0Length=Length;
        }
            Length =inputLength[i];
            currentSrcDst = inputSrcDst[i];
            Seq = inputSeq[i];
            Ack = inputAck[i];

            expectedResult = missing[i];
            calcMiss = expression();

            if(expectedResult>0 && calcMiss > 0){
              hits++;

        }
```

```
      else if (expectedResult ==0 && calcMiss <=0)
        hits++;
      else
              sum++;
     }//for i

      System.out.println("sum: "+sum);
      System.out.println("hits:"+hits);
      System.out.println("calcMiss: "+calcMiss);

   }
   public static double expression(){
  return ( (  (  (  ( Seq - M0Length )  -  ( M0Seq - 0.5915718 )  )  *
  (  ( Seq - -0.11322764 )  -  ( M0Ack * M0Seq )  )  )  *  (  (  ( Seq * Seq )
  -  ( -0.075646505 +  (  ( M0Ack * M0Length )  -  ( M0Ack * Seq )  )  )  )  *
  (  ( 0.56651837 - Ack )  *  ( Ack + M0Length )  )  )  ) );
}

public static double makeSt(){
  try{

  System.out.println("Opening tree.out");

     FileReader inputFileReader   = new FileReader("D:\\thesis\\data\\tree.in");

// Create Buffered/PrintWriter Objects
     BufferedReader inputStream   = new BufferedReader(inputFileReader);
     String outputFileName = "tree.out";
     FileWriter outputFileReader  = new FileWriter(outputFileName);
     PrintWriter outputStream  = new PrintWriter(outputFileReader);

     String inLine = null;
     String temp = new String();
     StringTokenizer st = new StringTokenizer(temp);
     int count = 0;
     Stack s = new Stack();
     Stack c = new Stack();
     String tmpSt1 = new String();
     String tmpSt2 = new String();

  while ((inLine = inputStream.readLine()) != null ) {
     st = new StringTokenizer(inLine, " \t ()");
     while(st.hasMoreTokens())
       s.push(st.nextToken());
  }
```

```
   while(!s.empty()){
     String tmp= (String)s.pop();
       if(tmp.equals("+") || tmp.equals("-") || tmp.equals("*")){
         tmpSt1=(String)c.pop();
         tmpSt2=(String)c.pop();

         c.push(new String( " ( "+ tmpSt1 + " " + tmp + " " + tmpSt2 + " ) " ));
       }
       else c.push(tmp);

   }
   //while(!c.empty())
     outputStream.print(c.pop());

     inputStream.close();
     outputStream.close();
     outputFileReader.close();
     inputFileReader.close();

   } catch (IOException e) {

                System.out.println("IOException:");
                e.printStackTrace();
       }
       return 0.0;
  }

  /**
   * Sole entry point to the class and application.
   * @param args Array of String arguments.
   */
  public static void main(String[] args) {

 if(!args[0].equals("makeSt")){
   setup(args[0]);
   evaluate();
 }
 else makeSt();
     System.out.println(args[0]);
  }

}
```

MultiValuedRegression.java

```
/*
Copyright 2006 by Sean Luke
Licensed under the Academic Free License version 3.0
See the file "LICENSE" for more information
*/

package ec.app.tcpmissing;
import ec.util.*;
import ec.*;
import ec.gp.*;
import ec.gp.koza.*;
import ec.simple.*;
import java.util.StringTokenizer;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class MultiValuedRegression extends GPProblem implements SimpleProblemForm
   {
   public static final String P_DATA = "data";
   public static final int SIZE_DATA = 36; // the number of packets read in from file

   public double[] missing= new double[SIZE_DATA];// = {0, 1, 2, 3, 4, 5, 6, 7, 8, 16};
   public double currentDeltaT=0;
   public double[] inputDeltaT= new double[SIZE_DATA];
   public double currentID=0;
   public double[] inputID= new double[SIZE_DATA];
   public double currentLength=0;
   public double[] inputLength= new double[SIZE_DATA];
   public double currentFlag=0;
   public double[] inputFlag= new double[SIZE_DATA];
   public double currentSrcDst=0;
   public double[] inputSrcDst= new double[SIZE_DATA];
   public double currentSeq=0;
   public double[] inputSeq= new double[SIZE_DATA];
   public double currentAck=0;
   public double[] inputAck= new double[SIZE_DATA];

   public DoubleData input;

   public Object clone()
      {
```

```
   MultiValuedRegression newobj = (MultiValuedRegression) (super.clone());
   newobj.input = (DoubleData)(input.clone());
   return newobj;
   }

public void setup(final EvolutionState state,
            final Parameter base)
   {
   // very important, remember this
   super.setup(state,base);

try{
//This is where I'm going to open the input file and read it in.
   System.out.println("Opening inputa.out...");

   FileReader inputFileReader   = new FileReader("D:\\thesis\\data\\default38.txt");

// Create Buffered/PrintWriter Objects
   BufferedReader inputStream   = new BufferedReader(inputFileReader);

   String inLine = null;
   String temp = null;
  StringTokenizer st = new StringTokenizer(temp=new String());
  int count = 0;

while ((inLine = inputStream.readLine()) != null && count < SIZE_DATA) {
    st = new StringTokenizer(inLine, " \t");
    missing[count]=(new Double(st.nextToken())).doubleValue();
    //System.out.print("missing: "+missing[count]);
    inputDeltaT[count] = (new Double(st.nextToken())).doubleValue();
    inputID[count] = (new Double(st.nextToken())).doubleValue();
    inputLength[count] = (new Double(st.nextToken())).doubleValue();
    inputSrcDst[count] = (new Double(st.nextToken())).doubleValue();
    temp = st.nextToken();
    if(temp.equalsIgnoreCase("S"))
      inputFlag[count]=0;
    else if(temp.equalsIgnoreCase("F"))
      inputFlag[count]=1;
    else if(temp.equalsIgnoreCase("R"))
      inputFlag[count]=2;
    else //if(temp.equalsIgnoreCase("P"))
      inputFlag[count]=3;
    //else inputFlag[count]=-1;
    inputSeq[count] = (new Double(st.nextToken())).doubleValue();
    inputAck[count] = (new Double(st.nextToken())).doubleValue();
```

```
      //System.out.print(" ack: "+inputAck[count]);
      count++;
      //System.out.println("count: "+count);

}//while
   } catch (IOException e) {

        System.out.println("IOException: "+e);
        e.printStackTrace();

   }

   // set up our input -- don't want to use the default base, it's unsafe here
   input = (DoubleData) state.parameters.getInstanceForParameterEq(
        base.push(P_DATA), null, DoubleData.class);
   input.setup(state,base.push(P_DATA));
   }

public void evaluate(final EvolutionState state,
                final Individual ind,
                final int threadnum)
   {
   if (!ind.evaluated)  // don't bother reevaluating
        {
        int hits = 0;
        double sum = 0.0;
        double expectedResult;
        double result;

      for(int i =0; i < SIZE_DATA; i++){
              currentDeltaT = inputDeltaT[i];
              currentID = inputID[i];
              currentFlag = inputFlag[i];

              if(currentSrcDst == 0){
                M0Seq.m0Seq = currentSeq;
                M0Ack.m0Ack = currentAck;
                M0Length.m0Length=currentLength;
        }else {
          M1Seq.m1Seq = currentSeq;
          M1Ack.m1Ack = currentAck;
                M1Length.m1Length=currentLength;
        }
              currentLength =inputLength[i];
              currentSrcDst = inputSrcDst[i];
```

```
            currentSeq = inputSeq[i];
            currentAck = inputAck[i];

            expectedResult = missing[i];
            ((GPIndividual)ind).trees[0].child.eval(
               state,threadnum,input,stack,((GPIndividual)ind),this);

            if(expectedResult>0 && input.x > 0){
              hits++;

            }
        else if (expectedResult ==0 && input.x <=0)
          hits++;
        else
              sum++;

      }//for i

        // the fitness better be KozaFitness! and it better be valid range
        if(Math.abs(sum)>Float.MAX_VALUE)
          sum = (Float.MAX_VALUE);
        KozaFitness f = ((KozaFitness)ind.fitness);
        f.setStandardizedFitness(state,(float)sum);
        f.hits = hits;
        ind.evaluated = true;
        }
    }
}
```

tcpmissing.params

```
# Copyright 2006 by Sean Luke and George Mason University
# Licensed under the Academic Free License version 3.0
# See the file "LICENSE" for more information

parent.0 = ../../gp/koza/koza.params
generations = 50
#777677
seed.0 =        time
pop.subpop.0.size =     10240

# the next four items are already defined in koza.params, but we
# put them here to be clear.

# We have one function set, of class GPFunctionSet
gp.fs.size = 1
gp.fs.0 = ec.gp.GPFunctionSet
# We'll call the function set "f0".  It uses the default GPFuncInfo class
gp.fs.0.name = f0
gp.fs.0.info = ec.gp.GPFuncInfo

# We have ten nodes in the function/dataset.  They are:
gp.fs.0.size = 10

gp.fs.0.func.0 = ec.app.tcpmissing.Length
gp.fs.0.func.0.nc = nc0
gp.fs.0.func.1 = ec.app.tcpmissing.RegERC
gp.fs.0.func.1.nc = nc0

#MEMORY 0 NODES
gp.fs.0.func.2 = ec.app.tcpmissing.M0Ack
gp.fs.0.func.2.nc = nc0
gp.fs.0.func.3 = ec.app.tcpmissing.M0Seq
gp.fs.0.func.3.nc = nc0
gp.fs.0.func.4 = ec.app.tcpmissing.M0Length
gp.fs.0.func.4.nc = nc0

gp.fs.0.func.5 = ec.app.tcpmissing.Seq
gp.fs.0.func.5.nc = nc0
gp.fs.0.func.6 = ec.app.tcpmissing.Ack
gp.fs.0.func.6.nc = nc0
gp.fs.0.func.7 = ec.app.tcpmissing.Add
gp.fs.0.func.7.nc = nc2
gp.fs.0.func.8 = ec.app.tcpmissing.Sub
gp.fs.0.func.8.nc = nc2
gp.fs.0.func.9 = ec.app.tcpmissing.Mul
gp.fs.0.func.9.nc = nc2


#Downsized Nodes, possible reentry in future implementations
#gp.fs.0.func.0 = ec.app.tcpmissing.DeltaT
#gp.fs.0.func.0.nc = nc0
#gp.fs.0.func.1 = ec.app.tcpmissing.ID
#gp.fs.0.func.1.nc = nc0
#gp.fs.0.func.4 = ec.app.tcpmissing.Flag
#gp.fs.0.func.4.nc = nc4
#gp.fs.0.func.2 = ec.app.tcpmissing.SrcDst
#gp.fs.0.func.2.nc = nc2

eval.problem = ec.app.tcpmissing.MultiValuedRegression
eval.problem.data = ec.app.tcpmissing.DoubleData
# The following should almost *always* be the same as eval.problem.data
# For those who are interested, it defines the data object used internally
# inside ADF stack contexts
eval.problem.stack.context.data = ec.app.tcpmissing.DoubleData
```

Ack.java

```
/*
Copyright 2006 by Sean Luke
Licensed under the Academic Free License version 3.0
See the file "LICENSE" for more information
*/


package ec.app.tcpmissing;
import ec.*;
import ec.gp.*;
import ec.util.*;

public class Ack extends GPNode
  {
  public String toString() { return "Ack"; }

  public void checkConstraints(final EvolutionState state,
                               final int tree,
                               final GPIndividual typicalIndividual,
                               final Parameter individualBase)
    {
    super.checkConstraints(state,tree,typicalIndividual,individualBase);
    if (children.length!=0)
      state.output.error("Incorrect number of children for node " +
                         toStringForError() + " at " +
                         individualBase);
    }

  public void eval(final EvolutionState state,
                   final int thread,
                   final GPData input,
                   final ADFStack stack,
                   final GPIndividual individual,
                   final Problem problem)
    {
    DoubleData rd = ((DoubleData)(input));
    rd.x = ((MultiValuedRegression)problem).currentAck;
    }
  }
```

out.stat

```
Generation 0
================

Subpopulation 0
----------------
Avg Nodes: 21.5955078125
Nodes/tree: [21.5955078125]
Avg Depth: 3.8587890625
Depth/tree: [3.8587890625]
Mean fitness raw: 19.439648 adjusted: 0.061916064 hits: 16.5603515625

Best Individual of Generation:
Evaluated: true
Fitness: Raw=1.0 Adjusted=0.5 Hits=35
Tree 0:
 (* (* (* (* (- M0Seq M0Ack) (+ Ack Ack))
    (+ (- M0Ack M0Ack) (- -0.35969096 M0Ack)))
    (* (- (+ Length Ack) (* 0.66489494 M0Ack))
       (- (- -0.34293437 M0Length) (* Ack Seq))))
    (* (* (+ (+ 0.80191904 -0.15020664) (- Seq
       Length)) (- (+ M0Length M0Seq) (- Seq -0.29533693)))
       (- (+ (+ Ack M0Length) (* Seq M0Ack)) (+ (+ M0Ack M0Length) (- Length Seq)))))

Generation 1
================

Subpopulation 0
----------------
Avg Nodes: 24.4669921875
Nodes/tree: [24.4669921875]
Avg Depth: 4.32158203125
Depth/tree: [4.32158203125]
Mean fitness raw: 14.832422 adjusted: 0.079840675 hits: 21.167578125

Best Individual of Generation:
Evaluated: true
Fitness: Raw=1.0 Adjusted=0.5 Hits=35
Tree 0:
 (* (* (* (* (- M0Seq M0Ack) (+ Ack Ack))
    (+ (- M0Ack M0Ack) (- -0.35969096 M0Ack)))
    (* (- (+ Length Ack) (* 0.66489494 M0Ack))
       (- (- -0.34293437 M0Length) (* Ack Seq))))
    (* (* (+ (+ 0.80191904 -0.15020664) (- Seq
       Length)) (- (+ M0Length M0Seq) (- Seq -0.29533693)))
       (- (+ (+ Ack M0Length) (* Seq M0Ack)) (+ (+ M0Ack M0Length) (- Length Seq)))))

Generation 2
================

Subpopulation 0
----------------
Avg Nodes: 32.061328125
Nodes/tree: [32.061328125]
Avg Depth: 5.130078125
Depth/tree: [5.130078125]
Mean fitness raw: 14.019824 adjusted: 0.084965155 hits: 21.98017578125

Best Individual of Generation:
Evaluated: true
Fitness: Raw=0.0 Adjusted=1.0 Hits=36
Tree 0:
 (+ (* (+ Seq M0Length) (* -0.99107873 M0Length))
    (* (- Seq M0Length) (- Seq M0Seq)))

Final Statistics
================
Total Individuals Evaluated: 30720

Best Individual of Run:
Evaluated: true
Fitness: Raw=0.0 Adjusted=1.0 Hits=36
Tree 0:
 (+ (* (+ Seq M0Length) (* -0.99107873 M0Length))
    (* (- Seq M0Length) (- Seq M0Seq)))

Timings
=======
Initialization: 0.501 secs total, 221138 nodes, 441393.22 nodes/sec
Evaluating: 1.101 secs total, 799988 nodes, 726601.3 nodes/sec
Breeding: 1.032 secs total, 578850 nodes, 560901.2 nodes/sec

Memory Usage
==============
Initialization: 12225.477 KB total, 221138 nodes, 18.088293 nodes/KB
Evaluating: 0.0 KB total, 799988 nodes, Infinity nodes/KB
Breeding: 24170.758 KB total, 578850 nodes, 23.94836 nodes/KB
```