5-2020

An alternative approach to malware research

Christopher James May
University of Northern Iowa

AN ALTERNATIVE APPROACH TO MALWARE RESEARCH

A Thesis Submitted

in Partial Fulfillment

of the Requirements for the Designation

University Honors

Christopher James May

University of Northern Iowa

May 2020

This Study by: C.J. May

Entitled: An Alternative Approach to Malware Research

has been approved as meeting the thesis or project requirement for the Designation University Honors

_____	_____
Date	Dr. Eugene Wallingford, Honors Thesis Advisor, Computer Science Dept.
_____	_____
Date	Dr. Jessica Moon, Director, University Honors Program

An Alternative Approach to Malware Research

C.J. May
Department of Computer Science
University of Northern Iowa

***Abstract* --- Current antivirus programs have design flaws that allow malware to bypass detection. Despite this, malicious parties are usually the ones to find and exploit these flaws before they can be fixed. Therefore, a more proactive approach to malware research should become the new standard. To that end, a new programming language will be designed and created that sheds light on a couple of design flaws in current antivirus models. Fundamentally, antivirus programs have trouble detecting interpreted languages. In addition, it is suspected that antivirus programs are unable to detect an unknown programming language that is injected into another file thus creating polyglot code. The Jaws programming language has been designed to exploit both of these weaknesses, and its implementation proves that such a language can exist.**

***Index Terms* --- artificial intelligence (AI), instruction modification parameter (IMP)**

I. Purpose

The purpose of this research is to prove that even the most advanced antivirus programs have design gaps in detecting emerging threats, and that a more proactive approach to malware research must become the new standard. The proactive approach to be tested in the hypothesis should be thought of as a kind of vulnerability research for antivirus programs. The exploited vulnerability in this research is purported to shed light on an unintended design flaw in antivirus software.

II. Literature/Source Review

To understand vulnerability research and why it is done, we must first explore the topics of malware and antivirus software. In November of 1988, the world had a wake up call about the potential severity of malicious programs. A graduate student at Cornell University, Robert Morris, wrote a program later called the Morris Worm that ended up bringing down almost the entire internet in just 24 hours [3]. This was before the world wide web, so the chaos was not quite as wide-spread as it would be today. Nevertheless, there was estimated to be millions of dollars in damages. The incident caused the whole world to understand how vulnerable computers had become. Even today, preventing the infection of malicious software (malware) largely remains a game of cat-and-mouse.

Since the Morris Worm brought down the whole internet, people have better understood the power of malware. Malicious hackers, hacktivists, and even nation states now develop malware for the purpose of manipulating computers for financial or political gain. For example, in 2010 the United States and Israel launched a cyber attack called Stuxnet that destroyed Iran's nuclear facilities in an effort to stymie their efforts in developing nuclear weapons [5]. That attack was also a type of worm, and it was able to be spread by USB drives. The most interesting part of the Stuxnet attack was that it proved that malware can not only affect cyber space; malware can impact physical space as well.

Other than worms, there have been many different types of malware that have emerged [10]. Adware delivers mass ads to infected computers. Ransomware holds a system's files captive via encryption and only returns them in exchange for payment. Spyware records and exfiltrates a user's activity without them knowing. Trojan Horses disguise themselves as a normal program but do malicious things in the background. There are other kinds of malware as well, and all have to do with manipulating infected computers to do the attacker's bidding.

As the prevalence of malware has increased, so have measures to detect and prevent malware. Antivirus programs are an example of software designed to combat malware. Antivirus programs scan the files on a computer and evaluate whether or not they are malicious.

If a file is found to be malicious, it is deleted or quarantined. Of the most popular antivirus programs [9], there are two main categories of antivirus software: signature-based and artificial-intelligence-based.

Signature-based antivirus has been around for a long time because it has proven to be mostly effective [8]. Signature-based antivirus programs keep definitions, or "hashes," of known malware and periodically scan a computer's file system looking for a matching hash [6]. If a file matches known malware, the antivirus program deletes or quarantines the file. Signature-based antivirus excels at finding the most common types of malware that are currently wreaking havoc in the wild.

There are several weaknesses to the signature-based model, however [6]. New strains of malware go completely undetected since they are not yet in the antivirus's definition list. In addition, the sheer volume of unique malware can make a definition list quickly grow too big in size, and most signature-based antivirus have to leave out many definitions for this reason. Polymorphic, or self-changing, malware is also very good at evading detection because it is able to change its signature every time it replicates. Although signature-based antivirus excels at stopping known malware, advances made by cyber criminals in malware design have been slowly making it less effective.

The weaknesses of signature-based antivirus have been mostly addressed in a newer, emerging type of antivirus based on behavior. At the front of the behavior-based antivirus industry is AI (artificial intelligence). AI-based antivirus [4] doesn't require large databases of malware hashes to look out for. Instead, the software contains a neural network that has been trained to positively identify files that *look like malware*. This not only removes the need for large definition files, but it also allows the antivirus software to identify new and polymorphic threats that somewhat resemble known malware.

Unfortunately, there are still some caveats to the current AI-based antivirus models. Some AI-based antivirus programs scan compiled binary files for strings, and use those strings to derive a malware score [2]. One vulnerability found in this model is to inject the malware with "happy" strings that would make the file look less suspicious. This vulnerability has been exploited by researchers against Cylance Protect, the leading vendor in the AI antivirus field [2]. Cylance fixed the vulnerability before it could be exploited in the wild because it was responsibly disclosed by the researchers.

Another way to evade AI-based antivirus is to write malicious code in a language that is not compiled to binary [7]. The difference between a compiled language and one that runs on a virtual machine should first be explained. At a high level, a computer is a complex machine that only interprets 1's and 0's (binary), and it uses those signals to perform calculations and tasks. When programmers write code, it has to be translated into binary before it can be run. A program that performs this translation is called a compiler. A compiler is a complex program. First, it inputs a text file written in a language, or code, defined for the compiler. Then, after performing lots of operations, the compiler outputs an executable program in the form of a binary file. A virtual machine is similar to a compiler, but it does not output the translated binary to an executable file. Instead, once a virtual machine interprets an instruction from the input, it executes it on behalf of the code.

Languages that run on virtual machines such as Java are an example of this. These types of programming languages are interpreted and immediately executed rather than compiled to binary and saved to an executable file which makes them harder or impossible for an antivirus program to detect [7]. Unpublished research done by my colleague Tony Nizzi concluded that the engine for any kind of antivirus software would need to include some form of the virtual machine that the code runs on in order to be able to detect it through static analysis. This would greatly increase the size of the antivirus program for each virtual machine included. In addition, less popular or new virtual machines would almost assuredly not be included.

III. Hypothesis

Both models of antivirus software have proven to be effective against the majority of malware; however, what would happen if a new kind of threat emerged that was designed specifically to evade both AI and signature-based antivirus? It would be better if this kind of threat were accounted for before it may be unleashed by a malicious party. The hypothesis for this thesis is that this kind of proactive approach to malware research would greatly benefit the antivirus industry. As shown by the Cylance Protect case, there are weaknesses in even the most advanced antivirus models that can be fixed [2]. To test the hypothesis, research must be done to design and develop a proof of concept malicious program that is able

to evade even the most advanced antivirus software. Then, mitigation techniques will be offered to show that improvements to current anti-malware models can be made.

IV. Methodology

The proof of concept program for this research must be capable of exploiting the weaknesses of both models of antivirus. Therefore, a virtual machine for a completely new language must be designed. For the proof of concept in this research, a new programming language called Jaws has been designed that runs on a custom virtual machine. Jaws is an esoteric, interpreted programming language that is based on another, called Whitespace [11], with added functionality. Jaws is an imperative, stack based language in which the only lexical tokens are the characters *Space*, *Tab*, and *New Line*. These characters, being invisible, are commonly called "whitespace". The name Jaws is an acronym (Just Another WhiteSpace), but the word itself was also intended to hold meaning because the code, being invisible to the human eye, is like a threat hidden beneath the surface. The reason for this is that Jaws was specifically designed to enable it to be used in polyglot code.

Polyglot code is a computer program or script contained in a single file, yet written in a valid form of multiple programming languages. Jaws interprets only whitespace characters while ignoring all other characters. Because of this, it is possible to easily create polyglot code by injecting Jaws into many types of files. This includes, but is not limited to: other programming languages, markup languages, text files, and image files. The next section of this document goes into detail about how Jaws code would be injected into various types of files without breaking either one's functionality.

The Jaws virtual machine has been designed to ignore any whitespace characters that are deemed to be not part of the Jaws program. This is made possible with the definition of a header and a footer that designate which areas of a file should be interpreted as Jaws code. In the lexer component of the Jaws virtual machine, character interpretation is stalled until the Jaws header has been scanned from the file. After the header has been scanned, any whitespace characters are then interpreted as Jaws code. If the footer is scanned after any complete statement, the lexer will again stall interpretation until another header has been read. The signal for end-of-program is interpreted differently than the footer, which allows Jaws code to be broken up and scattered throughout different parts of a file.

Programming and markup languages that are not whitespace controlled such as Javascript, C/C++, Perl, and HTML/CSS can be injected with Jaws code very easily. These kinds of languages usually require an arbitrary number of whitespace characters between tokens, but do not have strict requirements on the type, amount, or arrangement. The only requirement for this type of code injection is that every instance of a whitespace character in the original code is replaced with a part of the Jaws code. This requirement is to prevent whitespace characters not part of the Jaws code from being interpreted by the Jaws virtual machine while it is interpreting.

Jaws can be injected into Python code, but the process is slightly more complex than it is for most other programming languages. Because spacing and indentation matter in Python code, Python requires a fixed pattern of whitespace characters within its code. It cannot be modified to replace spaces or tabs with an arbitrary amount of whitespace characters, nor can it prepend/append whitespace characters to a line of code. Whitespace can, however, be found in arbitrary arrangements on their own lines within Python code. This means that lines of Jaws code can be placed before or after lines of Python code without interfering with Python's interpreter. In addition, because of the implementation of the Jaws header and footer, the required whitespace in the Python code can be ignored by the Jaws lexer.

Files that consist of plain text in the form of unicode or ascii characters can be injected in the same way as non-whitespace-controlled programming languages. The only difference with plain text files is that you can technically place the whitespace characters from the Jaws code anywhere without breaking the 'functionality' of the text file. The requirement of removing all previously existing whitespace characters still exists, but they technically do not need to be replaced in the same location unless you want to retain the groupings of characters that form 'words'.

Jaws code can also be injected into image files because of the Jaws header and footer. Many image file types can be injected with another file, completely hiding the second file in the process. This is because image files such as JPEG, PNG, and GIF ignore all data following

their footer. Anything after the image footer is ignored by programs opening the file. Therefore, any type of file or language that is interpreted starting at a designated header can be placed below an image footer in the same file. Because the Jaws virtual machine can begin interpretation at a designated header, it can ignore any 'whitespace' found in the image binary and then begin interpretation at any time after. In fact, Jaws code can also be injected into an image file that has already been combined with another file such as a zip archive.

The ability of Jaws code to be injected into other files is the key trait that allows it to achieve a high level of obfuscation. Jaws can truly hide in plain sight. It is expected that automated static analysis techniques will be completely thrown off by the original file's type or language. Even humans performing manual analysis should be unlikely to catch the hidden Jaws code because it is invisible to the human eye. Additionally, one of the first things that is done when doing manual analysis on malicious source code is to clean up the whitespace and make it more readable. In the case of a polyglot file injected with Jaws code, this would only erase the evidence. This design choice in the proof of concept is to prove that even the most advanced malware detection techniques have serious gaps.

Another design choice in the proof of concept was to write the virtual machine in the C programming language so it can run on any architecture that C compiles to. Since C can be compiled to almost any architecture, the Jaws virtual machine can be run almost anywhere as well. This includes computers with x64 architecture (most laptops, desktops, and servers), ARM architecture (embedded systems and mobile phones), and even web browsers via WebAssembly. This opens up a lot of attack vectors that will have to be explored and evaluated separately, since malware prevention software looks different on all of these targets.

V. Implementation

The Jaws programming language had a lot of thought put into the design in order to meet the requirements set in the hypothesis. The final language specification for Jaws that is implemented in the Jaws virtual machine is shown in Figure 1 below:

Figure 1: Jaws Language Specification

Jaws Language Specification:

Lexical Tokens

The only lexical tokens in Jaws are *Space* (ASCII 32_{10}), *Tab*, (ASCII 9_{10}), and *Line Feed* (ASCII 10_{10}). The choice to use line feed only and not carriage return was to avoid DOS/Unix conversion problems.

Starting/Stopping Interpretation

Jaws code will only interpret whitespace tokens in the section of the file between the Jaws Header and Footer. There can be any number of such sections in the same file. This gives the Jaws interpreter the ability to start and stop interpretation any number of times until the End-of-Program statement is reached. The tokens that make up the Header and the Footer are identical, but the End-of-Program instruction is unique and signals the end of the program.

[LF][Tab][Space]	Header/Footer
[LF][LF][LF]	End-of-Program

Instruction Set

Each instruction consists of two parts: The Instruction Modification Parameter (IMP) and the command. The IMP describes what type of operation the command is. The command is interpreted based on which IMP preceded it, and it is then executed accordingly. The IMPs and their commands are listed below.

Instruction Modification Parameter (IMP)

The IMP is the first part of a Jaws instruction. The command following it will be interpreted differently depending on which IMP is selected. The chart below illustrates each IMP:

[Space][Space]	Stack Manipulation
[Space][Tab]	Arithmetic
[Tab][Tab]	Heap Access
[LF][Space]	Flow Control
[Tab][LF]	I/O Action
[Tab][Space]	I/O Control

Commands

The commands for each IMP are organized together. The character(s) for the command follow directly after the IMP's characters with no delimiter. Some commands require a parameter as a part of the instruction. In these cases, the parameter will immediately follow the command in the form of a binary number. [Space] represents 0, [Tab] represents 1, and a [LF] signals the end of the parameter. Read more on parameters below the commands.

Stack Manipulation (IMP: [Space][Space])

Stack manipulation is the most commonly used instruction type. There are four stack instructions.

[Space] (Parameter: Data)	Push a literal onto the stack
[LF][Space]	Duplicate the top item on the stack
[LF][Tab]	Swap the top two items on the stack
[LF][LF]	Discard the top item on the stack

Arithmetic (IMP: [Space][Tab])

Arithmetic commands operate on the top two items on the stack, and replace them with the result of the operation. The first item to be popped is considered to be to the left of the operator.

[Space][Space]	Addition
[Space][Tab]	Subtraction
[Space][LF]	Multiplication
[Tab][Space]	Integer Division

[Tab][Tab]	Modulo

Heap Access (IMP: [Tab][Tab])

Heap access commands look at the stack to find the address of items to be stored or retrieved. To store an item, push the address then the value and run the store command. To retrieve an item, push the address and run the retrieve command, which will replace the address at the top of the stack.

[Space]	Store
[Tab]	Retrieve

Flow Control (IMP: [LF][Space])

Flow control operations are also very common. Labels mark the targets of conditional and unconditional jumps as well as subroutines. Flow control operations allow high-level logic like loops, if-statements, and functions to be implemented.

[Space][Space] (Parameter: Label)	Mark a location in the program
[Space][Tab] (Parameter: Label)	Call a subroutine
[Space][LF] (Parameter: Label)	Jump unconditionally to a label
[Tab][Space] (Parameter: Label)	Jump to a label if the top of the stack is zero
[Tab][Tab] (Parameter: Label)	Jump to a label if the top of the stack is negative
[Tab][LF]	End a subroutine and jump back to caller

I/O Action (IMP: [Tab][LF])

We need to be able to interact with the user and the disk. There are I/O instructions for reading and writing numbers and individual characters.

[Space][Space]	Output the character at the top of the stack
[Space][Tab]	Output the number at the top of the stack
[Tab][Space]	Read a character and place it on the top of the stack
[Tab][Tab]	Read a number and place it on the top of the stack

I/O Control (IMP [Tab][Space])

We need to be able to read and write from the disk or to communicate over a network. To do that, we will change the I/O stream from standard in/out to a file.

[Space][Space]	Change I/O stream to a File -- get mode character and then file path from the stack
[Space][Tab] (Parameters: IP, Port)	Change I/O stream to TCP connection at IP, Port
[Tab][Space]	Change I/O to standard in/out

NOTES: File mode is one of 3 characters: r, w, or a. Each file mode's functionality is equivalent to C language's r+, w+, and a+ modes. File path is between '{ }' brackets and popped characters are arranged left-to-right (push characters onto the stack backwards so they are popped in order)

Command Parameters

Each parameter type is fixed-length. A binary data literal pushed onto the stack is either 32 bits (int) or 8 bits (char). At runtime, the type of data pushed onto the stack depends on the size of the parameter. Type checking is done before the data reaches the stack. where the data involved is explicitly declared by the language. Label parameters are 16 bits long, leaving room for 65,536 different labels. Network connection parameters are 48 bits long -- 32 bits to specify the IP address, followed by 16 bits to specify the port number.

End of Figure 1

Once the language was designed, the interpreter had to be written. I have decided to only explain the code at a high level rather than include the source in this thesis for two reasons. First, the source code contains many thousands of lines, and it would be too verbose to include in full. The second reason is due to the nature of the Jaws programming language. It may be ethically irresponsible to open-source the virtual machine before the threat it potentially poses is proven to be accounted for in antivirus software.

As mentioned in the methodology, the Jaws virtual machine was written in the C programming language. To aid in speeding up the project's development timeline, Flex and Bison [11] were used to generate C code for the parser. Flex and Bison are programs that generate a lexer and a parser (respectively) through the definition of a language's tokens and grammar. Once the language specification above was translated into valid Flex and Bison files, they generated the lexer and parser parts of the Jaws virtual machine in the form of C code.

Flex and Bison files are divided up into three sections. The first section is the control section. The control section includes things like C headers that will get placed in the generated file, as well as options for Flex or Bison when generating the lexer and parser. The Jaws virtual machine includes options for importing C libraries and keeping track of line numbers. The second section is specific to Flex or Bison. More on Jaws' implementation of the second section is found in the following paragraphs. The last section contains C code to be copied verbatim to the generated parser. In the Jaws virtual machine's source code, this section is for defining the 'main' function of the interpreter.

The first code that was written for the Jaws virtual machine was the Flex file that is used to generate the Jaws lexer. The Jaws Flex file only defines three tokens (Space, Tab, and Linefeed). Everything else is ignored as specified by a line containing the wildcard token for all other characters. All the lexer generated by Flex does is capture these three tokens and pass them on to the parser generated by Bison. The Jaws Flex file is relatively short, because there are only three tokens for the language and they are each only one character.

Next, the Jaws Bison file was written for the generation of the parser. A Bison file's second section consists of the formal grammar of the language with the tokens from the Flex lexer as terminals. Bison generates the parser from that formal grammar. The formal grammar for the Jaws programming language is defined in Figure 2 below:

Figure 2: Jaws Formal Grammar

jaws (start symbol)	⇒	<bodies> <last_body> | <last_body>
bodies	⇒	<bodies> <body> | <body>
body	⇒	<header> <instructions> <footer>
last_body	⇒	<header> <instructions> <end_program>

header	⇒	<extra_lines> LF TAB SPACE
		| LF TAB SPACE
footer	⇒	LF TAB SPACE
end_program	⇒	<end_instruction> <extra_lines>
		| <end_instruction>
end_instruction	⇒	LF LF LF
extra_lines	⇒	<extra_lines> <extra_line> | <extra_line>
extra_line	⇒	SPACE | TAB | LF
instructions	⇒	<instructions> <instruction> | <instruction>
instruction	⇒	<stack_manipulation> | <arithmetic> | <heap_access> | <flow_control> | <io_action> | <io_control>
stack_manipulation	⇒	SPACE SPACE <stack_command>
arithmetic	⇒	SPACE TAB <arith_command>
heap_access	⇒	TAB TAB <heap_command>
flow_control	⇒	LF SPACE <flow_command>
io_action	⇒	TAB LF <io_action_command>
io_control	⇒	TAB SPACE <io_control_command>
stack_command	⇒	<stack_push> | <stack_duplicate> | <stack_swap> | <stack_discard>
arith_command	⇒	<addition> | <subtraction> | <multiplication> | <integer_division> | <modulo>
heap_command	⇒	<heap_store> | <heap_retrieve>
flow_command	⇒	<new_label> | <call_subroutine> | <uncond_jump> | <jump_if_zero> | <jump_if_neg> | <end_subroutine>
io_action_command	⇒	<output_char> | <output_int> | <read_char> | <read_int>

io_control_command	⇒	<stream_file> | <stream_net> | <stream_stdio>
stack_push	⇒	SPACE <number>
stack_duplicate	⇒	LF SPACE
stack_swap	⇒	LF TAB
stack_discard	⇒	LF LF
addition	⇒	SPACE SPACE
subtraction	⇒	SPACE TAB
multiplication	⇒	SPACE LF
integer_division	⇒	TAB SPACE
modulo	⇒	TAB TAB
heap_store	⇒	SPACE
heap_retrieve	⇒	TAB
new_label	⇒	SPACE SPACE <label>
call_subroutine	⇒	SPACE TAB <label>
uncond_jump	⇒	SPACE LF <label>
jump_if_zero	⇒	TAB SPACE <label>
jump_if_neg	⇒	TAB TAB <label>
end_subroutine	⇒	TAB LF
output_char	⇒	SPACE SPACE
output_int	⇒	SPACE TAB
read_char	⇒	TAB SPACE
read_int	⇒	TAB TAB
stream_file	⇒	SPACE SPACE
stream_net	⇒	SPACE TAB <ip> <port>
stream_stdio	⇒	TAB SPACE
number	⇒	<bits> LF
label	⇒	<bits> LF
bits	⇒	<bits> <bit> | <bit>
ip	⇒	<octet> <octet> <octet> <octet>
octet	⇒	<bit> <bit> <bit> <bit> <bit> <bit> <bit> <bit>
port	⇒	<octet> <octet> LF
bit	⇒	SPACE | TAB

End of Figure 2

Next, the capability for Jaws code to be injected into whitespace-controlled programming languages like Python was added to the Jaws virtual machine. As mentioned earlier, in the case of a whitespace-controlled language like Python you can have arbitrary whitespace on lines separate from Python code. However, the whitespace characters in the lines of Python would break the functionality of the Jaws lines. For cases like that, a feature was added to the Jaws virtual machine that enables it to start and stop interpretation of whitespace characters using header and footer statements. Once the rest of the formal grammar was parsing correctly, the header and footer statement functionality was added into the grammar in the Jaws Bison File.

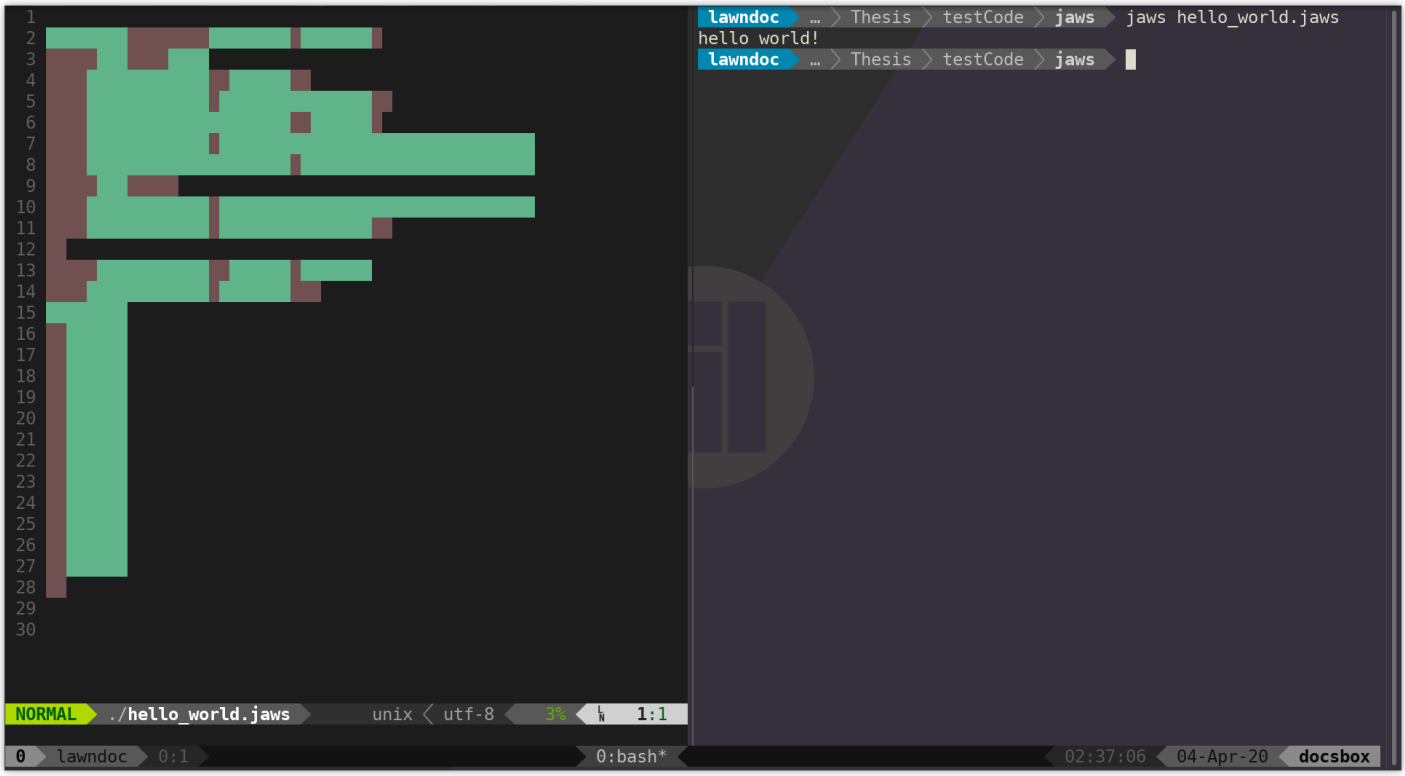
The final part of the Jaws virtual machine that had to be written was the runtime system that would execute the code. The Jaws runtime system is implemented in pure C code. Having only written a single traditional compiler before this project, creating a runtime system for the Jaws virtual machine was uncharted territory for me. I initially had some preconceived notions of how to do it that I quickly figured out wouldn't work. After brainstorming and consulting with my research advisor, Dr. Eugene Wallingford, I decided the best plan of action was to create multiple data structures to represent the program. There would be a data structure for a stack, a heap, an instruction, and a program (which contains an array of instructions). At the recommendation of Dr. Wallingford, I also created a data structure for a jump table that could help the runtime system execute jumps. Various functions to interact with each of these data structures were written. Lastly, semantic actions were added to the Bison file's formal grammar to construct some of these structures during the parsing of Jaws code.

Next, functions were created to represent each type of instruction defined in the language specification. A pointer to the corresponding instruction function is included in the instruction data structure once it is identified and created. As the program data structure iterates through the program, it simply keeps track of an instruction pointer and calls the instructions' functions as they come. Flow control operations may modify the instruction pointer, allowing for instructions to be called more than once.

Lastly, a lot of error-checking code was added to both the parser and the runtime system. It is important for a compiler or interpreter to provide informative error messages. Jaws error messages display the type of error, exactly why it happened, which instruction type caused the error, and what line of code the error occurred on. Once error-checking was done being added, the Jaws virtual machine was finished.

The complete Jaws virtual machine can indeed run 'invisible code' that is completely composed of Spaces, Tabs, and Linefeeds. Below, Figure 3 shows the traditional 'Hello World' program on the left written in Jaws (with spaces and tabs highlighted for visual aid). The output of the program is displayed on the right.

Figure 3: Hello World Jaws Program



As mentioned earlier, it was also necessary to create a visible language that can compile into Jaws code. This enables anyone to write and debug code much more quickly when developing Jaws programs. The name of the visible Jaws code is “Fin”. The Fin language has a traditional compiler that outputs Jaws code rather than executing a runtime system. For the Fin compiler, Flex and Bison were used again to generate the lexer and parser. That allowed me to reuse a lot of the formal grammar I defined in Jaws’s Bison file.

The Flex file for the Fin compiler was quite a bit more work than it was for Jaws. Jaws only has three tokens, but a readable version of Fin called for a separate token for each instruction. For example, ‘add’, ‘sub’, ‘mult’, and ‘div’ are all individual tokens, rather than each one being made of the same three characters.

Another thing that added complexity to the Fin lexer was the addition of literals as tokens. In Jaws, instruction parameters such as numbers and IP addresses were simply defined in binary, with 1 being a Tab and 0 being a Space. However, in order to write code more easily, functionality was added to Fin to be able to write things like “2048”, “C”, and “127.0.0.1:22” as parameters. To do this, regular expressions were defined for each literal. Other than the number of tokens and the regular expressions for the literals, there wasn’t anything else that needed to be modified in the Fin Flex file.

A lot of the existing Jaws grammar was able to be copied to the Fin Bison file due to the two languages having the same syntactic structure. The tokens coming in from the lexer had to be redefined, and their corresponding terminal symbols had to be replaced in the grammar. Other than that, though, the formal grammar remained mostly the same. Once all the tokens were parsing correctly, code generation was the final part of the Fin compiler.

The code generation for the Fin compiler ended up being fairly simple. First, a Jaws output file to write to was declared in the main function of the parser. Then semantic actions were placed after each terminal symbol in the Bison grammar. Each semantic action writes the corresponding Jaws invisible character combination to the output file. Once that was complete, some polish was added to the Fin compiler. Command line arguments for specifying the output file and suppressing visible annotation of the Jaws code were added as options.

Additionally, makefiles and man pages were added to both the Fin compiler and the Jaws virtual machine to aid in ease of distribution.

VI. Expected Findings & Conclusion

Unfortunately, the goal of testing the hypothesis was not able to be completed before the submission of this thesis due to the time constraints that one semester affords and the amount of time it takes to write an interpreter and compiler. I was not able to do enough testing of malware written in Jaws code to be able to present any findings. I was, however, able to prove that a polyglot, interpreted programming language can exist. The Jaws programming language is designed to exploit every weakness of current antivirus models. The next steps in this research are to develop malware using the Jaws programming language and to run various tests against antivirus software to see if malicious code written in Jaws is able to be detected. Once proper testing is able to occur, it is expected that the proof of concept malware is able to evade all leading antivirus programs by exploiting an unintended weakness in the overall design.

In addition to the expected findings, it is important that design changes can be offered for current antivirus models that would mitigate the threat developed in the proof of concept. Currently, the design change for antivirus software that would be suggested is to add functionality to detect unknown interpreters. This would enable the antivirus to flag programs as suspicious. However, because an interpreter is not inherently malicious, this would still require laborious human analysis and is an inelegant solution in my opinion. A better solution may be quite complex. Therefore, additional research may also be needed in that case to ensure that the threat is adequately mitigated.

If the expected findings hold true, a programming language like Jaws would effectively render all antivirus software useless. If existing or future malware were to be rewritten in Jaws, they would be able to run without detection on any computer. The implications of this would be extrememly detrimental to the security of all computers. Private research like this is important in taking the advantage over malicious cyber adversaries because it allows us to stay ahead of the bad guys instead of cleaning up a mess when they discover something like Jaws. This alternative approach to malware research shows that anti-malware models can be improved through responsible development and disclosure of new types of malware and evasion techniques.

References

[1] A. A. Aaby, Compiler Construction Using Flex and Bison. College Place, WA: Walla Walla College, 2004.

[2] CERT Coordination Center. “Cylance Antivirus Products Susceptible to Concatenation Bypass”. Carnegie Mellon University, 1 Aug. 2019. Available: https://kb.cert.org/vuls/id/489481/

[3] FBI. “The Morris Worm.” *FBI*, 2-Nov-2018. Available: https://www.fbi.gov/news/stories/morris-worm-30-years-since-first-major-attack-on-internet-110218

[4] I. Firdausi, C. Lim, A. Erwin and A. S. Nugroho, "Analysis of Machine learning Techniques Used in Behavior-Based Malware Detection," 2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies, Jakarta, 2010, pp. 201-203.

[5] J. P. Farwell & Rafal Rohozinski. “Stuxnet and the Future of Cyber War”. *Survival*, Vol 53, Issue 1, pg. 23-40. 2011.

[6] J. Scott, “Signature Based Malware Detection is Dead,” Institute for Critical Infrastructure Technology, Feb-2017.

[7] K. L. Russo, “Obfuscation and Polymorphism in Interpreted Code,” SANS Institute, 10-Feb-2017.

[8] O. Sukwong, H. Kim, and J. Hoe, “Commercial Antivirus Software Effectiveness: An Empirical Study,” Computer, vol. 44, no. 3, pp. 63–70, Mar. 2011.

[9] OPSWAT, “Windows Anti-malware Market Share Report,” OPSWAT MetaDefender, 28-Feb-2020. [Online]. Available: https://metadefender.opswat.com/reports/anti-malware-market-share?date=2020-02-28.

[10] R. A. Grimes, “9 types of malware and how to recognize them,” *CSO Online*, 01-May-2019. [Online]. Available: https://www.csoonline.com/article/2615925/security-your-quick-guide-to-malware-types.html.

[11] “Whitespace tutorial”. *Web Archive*, 8-Nov-2015. Available: https://web.archive.org/web/20151108084710/http://compsoc.dur.ac.uk/whitespace/tutorial.html