Binary Filtering and Analysis: An Overview of the Tools and Processes that Provide Pivotal Reverse Engineering Information

By,
Ethan Benkel and Austin Peterson

Introduction and Purpose

At the start of every reverse software engineering endeavor, we are often met with the immediate problem of a lack of crucial information. What is the functionality of this binary executable? Is this executable safe to run? What is the general control flow of the program? How much of the code in this binary is actually relevant to what we are trying to accomplish? All of these questions need to be answered before we proceed to reverse engineer any software laid before us. Thankfully, with the aid of binary filtering and analysis techniques, as well as automated tools, these questions are much easier to answer than originally perceived. In this report, we intend to conduct a research survey that will explain and compare various binary filtering and analysis processes, as well as specific automated tools. In regard to processes and tools, we aim to explore those not yet detailed in class, while still comparing them to tools we have utilized, such as Ghidra. We also hope to assess the strengths and weaknesses of these processes and tools, as well as their overall importance, and ultimately deduce which tools are the most useful in a real-world setting.

Static Binary Filtering and Analysis Processes (Obfuscation)

Static binary filters have long been used to extract vital parts of machine code for the purpose of developing a control flow diagram, however this is by no means the only purpose of such filters. Binary filters can also be used to reveal obfuscated code regions, detect malware and viruses, and even conduct type-state checking of the machine code (Hanov, 1). Obfuscated, or unintelligible code is becoming an increasingly pertinent issue in the realm of cybersecurity and reverse software engineering. In a classroom, students see that a disassembly of binaries usually leads to a list of assembly commands that are fairly easy to organize and determine the complete functionality of. This is certainly not the case in real-world practice. Some binaries are purposefully obfuscated to prevent hackers from determining a piece of code's purpose and control flow. This obfuscation usually comes in the form of multiple instructions attempting to hide, or overlap with pivotal branch instructions. Predicates determining the control flow of instructions can also be made opaque, which causes the control flow of the program to be nearly impossible to decipher, as predicates determine when the ability to alter the architectural state of the system is granted to certain instructions in the program.

To combat the obfuscation of binary executables, we can layout a simple deobfuscator. The process behind such a device is primarily based on statistical observation rather than some advanced search algorithm hunting down branch instructions, which is usually utilized in most dynamic binary filters. Contiguous blocks of statements ending at a branching instruction become nodes in the control flow diagram, or CFD. Note that the CFD may contain blocks that overlap in memory. At this point in the deobfuscator's design, the designer claims that this is allowed. Secondly, for each branch statement, the source and destination nodes are connected. The destination node may be split, if the destination address is not at the start of the node. At this point in the algorithm, the CFD is a superset of the real control flow graph, or diagram of the procedure, because some of the blocks are overlapping. In the next step, these conflicts are resolved in a number of ways. Real basic blocks of instructions are assumed to be more tightly integrated into the control flow than spurious ones. If two basic blocks conflict, the block that is more tightly connected to other blocks is chosen to remain in the mapped out program control flow. If

there are remaining conflicting basic blocks, they are eliminated randomly (Kruegel, 5). This algorithm is still being worked on in the future work of the designer, further showing how the development of binary filtering techniques remains prominent in the realm of reverse software engineering.

Static Binary Filtering and Analysis Processes (Malware Detection)

Filtering of obfuscated binary code, also relates to malware detection, as obfuscation, aside from being used to improve the security and decrease understandability of code, is also used to hide malicious code. There are a plethora of techniques utilized by malware designers to hide their malicious code within the binaries of a program. The first of which is to simply insert multiple nop, or no operation instructions into the program binaries. Another technique involves the rearranging of instructions. This method assumes that the instructions being transposed are not dependent on the order of their execution. Similarly to this method, a virus can also perform instruction substitution where a single instruction is replaced with multiple instructions that ultimately perform the same functionality as the single replaced instruction. Lastly and most insidiously, a virus will sometimes obfuscate code by weaving instructions into the host program, causing the virus to execute concurrently with the host.

With all the methods a virus has to obfuscate a program binary, there is clearly a need for a binary filter that can detect and extract a sequence of malicious code instructions. A proposed static binary filter to do just this, consists of three phases of operation. The first and second steps of this filter are simply disassembly and code annotation. The third phase, the actual malware detection and filtering, is what we will focus on for the purpose of this paper. For a given virus, a malicious code automation is manually constructed. This automation is a sequence of abstract statements in the virus' code that do not include individual registers or machine code instructions, and are simply assembly instructions such as move and pop. Once the virus code is reduced to this form, the assembly code is converted to a high level abstraction. Now all that is left to do is determine whether any path in the control flow of the targeted program matches the path through the malicious code automation. This is done by viewing the control flow graph as a finite state machine that produces a regular language (Hanov, 3-6). Ultimately, if language produced by the malicious code automation, and the language of the targeted program share any common elements, one can infer that at least one path in the control flow of the program is malicious code.

Static Binary Filtering and Analysis Processes (Type-State Analysis)

Another facet of binary filtering deals with type-state analysis of machine code. Type-state analysis helps us determine aspects of the code, such as whether a particular variable in a program is always initialized before being read from. Type-state analysis essentially tells us how and what rules are assigned to variables in a program. In a higher level language example, this might come in the form of a lock function that can only be used on objects that aren't already locked. This analysis can be easily applied to machine code for the purpose of relating it to the concept of binary filtering.

In practice, it is vital that any machine code being run is verified to be safe and malware free. Binary type-state analysis helps us accomplish this by statically checking for unaligned memory accesses, null pointer dereferences, and array bounds accesses before we run any machine code whose security we haven't fully verified. From our research survey, it is likely this kind of analysis would be implemented as some type of plugin distributed to a host program. The analysis would consist of four major phases named preparation, type-state propagation, annotation, and local and global verification. Before moving forward, it is important that the terms of type-state, access policy, and invocation are clearly defined as these terms all pertain to vital attributes of registers being utilized by the program, and its associated variables that we wish to analyze. Type-state is simply the initial type-state of each variable found in the machine code. The access policy controls how the variables may be used, and is composed of the region of memory the

variable encompasses, the type of variable such as an array, or string, and finally, the access field, which determines if a variable is read, write, or read-write. Lastly, the invocation tells us what the initial state of the registers are.

Now let's discuss the actual stages of type-state analysis as it relates to machine code and binary filtering. The first stage, preparation is simply the gathering and consolidation of the type-state, access policy, and invocation of all variables in the host program. This information is used to create initial constraints on variables and registers. For example, a certain register may only be allowed to hold array variables. The second stage, type-state propagation, deals with abstractly interpreting statements within the host program and making copies of the memory state, and type-state for each variable, register, or memory location at pivotal points in the program's control flow. The third stage, annotation, creates assertions and preconditions for each program statement. For instance, if a variable indexes an array, a precondition must be applied to this variable, so that it always lies within the bounds of the array it is associated with.

Local preconditions can be verified with type-state information, while global preconditions may require further code analysis. The fourth and final stage, verification, simply checks that both types of preconditions on the program statements are checked. An example of a local check would be to use the type-state propagation information to check if a variable is initialized at the proper time in the program. A global precondition check, perhaps involving an array bound, would require a range analysis of the values in the pertinent registers storing the array. If all the preconditions of the program are verified, we can assume the program is safe to execute (Xu, 3).

Static binary filters evidently play a pivotal role in reverse software engineering and program analysis. Filtering algorithms don't just help us single out pivotal branch instructions for the sake of making a control flow diagram. They also help us single out obfuscated and malicious code. Furthermore, they extract vital type-state information from machine code for the purpose of verifying program integrity and security. While research in static binary filters continues to persist and remain relevant in the realm of cybersecurity, algorithms related to this topic will always have limitations in regard to scalability and precision. Whether it be the perpetual increase in the complexity of malware generation, or the reliance on a flow sensitive interprocedural algorithm, such as binary type-state analysis, there will always be some attribute of machine code that fails to be accounted for and renders the static filter to be incomprehensive, or in the worst case, completely obsolete.

Binary Filtering and Analysis Tools Introduction

Automated binary filtering and analysis of executable files is still considered to be a recent and promising field of research. Such research can help lead to innovation in software code verification, specifically in areas such as malware and legacy code. More importantly, this research can help provide a more accurate and comprehensive analysis of high-level language code developed from a decompilation analysis tool. Such a tool is one of the predominant features in Ghidra, a binary analysis tool my colleague and I utilized many times in our work related to reverse software engineering. Still, even a prominently used industry standard tool such as Ghidra cannot be assumed to always produce a comprehensive binary analysis without flaw, especially in regard to decompilation.

Aside from the theoretical challenges solved by the static methods we have previously discussed, innovation in binary filtering and analysis also suffers from practical issues as well. These come in the form of the immense amount of programming efforts required to implement an actual tool like Ghidra, as well as the existence of many different instruction set architectures (ISA), which adding support for can be a very tedious and time-consuming effort. These struggles are compounded by the fact that binary executables can be highly different from one another in regard to code semantics and general control flow. This fact will seemingly always present the issue of there being no single tool that is ideal for any

type of binary filtering or analysis, no matter how complex, or all encompassing the tools claim to be with their methods. A study by professors at the United States Military Academy at West Point, NY regarding the comparison of different reverse software engineering tools seem to have reached the same conclusion (Conti, 9). With all of this in mind, we feel it is of the utmost importance to explore more basic and concise frameworks, developed, or currently in development that actually implement the binary analysis and filtering techniques we've previously detailed. Through this exploration, we hope to develop a thorough understanding of how these frameworks function and are developed, and how eventually they may evolve into larger well-known tools such as Ghidra.

Binary Filtering and Analysis Tools (BINCOA - Simple Analysis)

The first automatic binary analysis and filtering implementation we researched was the Binary Code Analysis (BINCOA) framework. BINCOA's ultimate goal is to ease the development of binary code filters and analysers. It accomplishes this by providing an open formal model for executable binaries and other low-level programs in an XML format that provides easy exchanging of models, and minimal basic tool support. BINCOA was constructed around the Dynamic Bitvector Automata (DBA), which is a generic and very concise formal model for low-level binary executables. This essentially means that DBA only supports a small set of program instructions and can really only model common program architectures, as opposed to overly complex ones. However, DBA does provide a sufficient low-level formalism, which allows the automata to serve as a decent reference semantics of the executable binary being analyzed, or filtered (Bardin,1).

Branch instructions that relate to dynamic jumps and modifying the program call stack, as well as common binary executable elements of interest, such as instruction overlaps from obfuscation, and endianness, or instruction format can all be successfully filtered with the functionality provided by DBA (Bardin, 3). While BINCOA's reliance on DBA clearly limits the scalability of the tool in regard to supporting other instruction set architectures, as well as not having the ability to successfully filter complex binaries, it is very concise and easy to use. BINCOA may not be able to comprehensively analyze an executable containing self-modifying malware, but it can definitely retrieve pivotal branch instructions, and perform some basic deobfuscation.

BINCOA supports three major binary filtering and analysis functions. These include test data generation, control flow diagram recovery, and code simulation. Test data generation, in this case, is primarily used to predict the outcome and functionality of the code being analyzed, while avoiding any legitimate execution of malicious code. This is done through dynamic symbolic execution and a method known as bit-vector constraint solving. While we don't intend to focus on the inner-workings of symbolic execution in this paper, the bit-vector constraint method partially relates to the static binary analysis and filtering we discussed earlier aptly named type-state checking. After we know what a certain variable in memory is, for example, an integer type, we can apply certain constraints to it, such as what functions can access it, under what conditions it can be accessed, and what values it may store. Bit-vector constraints are simply the constraints placed on variables in regard to how many bits they may use in memory (Bordeaux, 2). Obviously some variables will have larger domains and much less strict bit constraints than others. Knowing these constraints and attributes of a binary's variables make it much easier to generate test data from the code, because the variable's purpose and functionality can be more easily inferred and analyzed.

BINCOA's control flow diagram recovery runs off a refinement-based static analysis algorithm (Bardin, 4). This algorithm is actually very simplistic in nature and is very similar to how malicious code is filtered out of a binary. The targeted binary executable is broken up into multiple path expressions of control flow. Instead of comparing the language in each block of the control flow to blocks of malicious code, language in each path expression is compared to each other to ultimately deduce the general

importance and worth of each control flow block to the overall program functionality (Cyphert, 1-3). The value of each path expression can be measured quite differently depending on the algorithm implementation. BINCOA's provided documentation does not specify this attribute of their implementation, however we can likely assume it has something to do with how some control flow blocks are connected to more blocks than others. It would not be outlandish to assume that blocks that affect the most amount of paths in the control flow are the most pivotal to the executable's functionality, however this is merely our conjecture.

Lastly, BINCOA's code simulation relates mainly to its binary disassembly ability and code representation in the form of an XML. While the XML representation of the analyzed binary is simply to aid analysts in sharing their findings in a graphical form, the binary disassembly ability is pivotal to attempting any form of binary filtering and analysis. Every automated form of binary filtering and analysis my colleague and I have come across requires some form of code disassembly, either static, or dynamic. The instruction disassembly and encoding in BINCOA is all done through Insight, a platform that is primarily developed in C++. BINCOA, also supports easy annotation of code, both automatic and manually, which is another feature we have found to be very necessary in any tool used for binary filtering and reverse software engineering in general. This is simply because they are both incremental processes that require taking notes at each pivotal step.

One final thing to note on BINCOA is that while it is very simplistic and is nowhere near the development stage to be on par with a tool such as Ghidra, it represents a key factor in how binary filtering and analysis tools are often implemented. This can bluntly be stated as a cooperation between different tools. There is a good reason BINCOA supports both test data generation and control flow diagram recovery. While these methods on their own are helpful to understanding an executable and finding important attributes in their own right, together they accomplish what the two alone can not. BINCOA's test data generation can receive an upper approximation of every set of jump instruction targets from the control flow graph recovery tool. This helps the test data generation tool to provide security coverage measures on each jump instruction to prevent potential malicious code in the binary executable from running during critical system certification efforts. This is a clear example of why binary filtering and analysis tools provide such a plethora and variety of tool types. These tools working together can provide a greater level of security and comprehensive analysis, which are often both necessary in industry and real world examples due to the growing threats of increased malware complexity and cyberattacks.

Binary Filtering and Analysis Tools (BitBlaze - Controlled Execution Analysis)

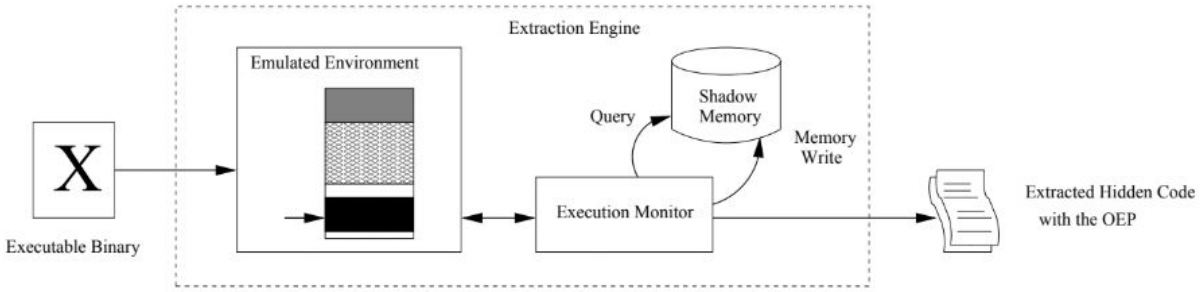
Continuing with our automated examples of binary filtering and analysis, we take a look at a platform called BitBlaze. The platform itself consists of three main components. Vine, a static analysis component, provides core utilities for common analysis and allows for symbolic execution. Next is the dynamic analysis component TEMU, which allows for the execution of binary files in a secure, emulated environment, as well as binary instrumentation and the extraction of semantic information. Lastly, Bitblaze uses Rudder, BitFuzz and Fuzzball which all serve as the symbolic exploration components (Song, 2). The most valuable aspect of this platform is widely considered to be TEMU, which allows for the execution of programs without the risk of negative effects from malicious files.

Where this system stands out is in the fact that the user can observe the effect a program has on multiple different processes instead of observing a generic localized view of the program's execution. Most examples of malware operate on a more widespread level, affecting multiple different user processes instead of just running independently. TEMU emulates an entire system, including the operating system and all of its components, where it then has the ability to observe every aspect of a program's execution at a very fine level. This proves to be a very valuable and often necessary feature for

security engineers. Additionally, this dynamic component is implemented with modules that aid in the analysis of binary files, including code unpacking, malware analysis and trace logging (Song, 5). Bitblaze has a variety of tools that are useful on their own, however, since it was developed nearly ten years ago, its greatest lasting contribution to security engineering is the base engine that it provided to many other developers.

Binary Filtering and Analysis Tools (Renovo - Packed Code Extraction)

One of the best examples of the use of BitBlaze's base engine has been the development of Renovo, a program able to extract hidden code from packed binary executables. A common anti-reversing technique is the packing of malware, which compresses and hides the program's code until it is unpacked during execution. This program, built on Bitblaze's dynamic component, uses TEMU's emulated monitor and environment to execute potentially malicious files and observe their effects. Renovo's methods are actually quite simple.



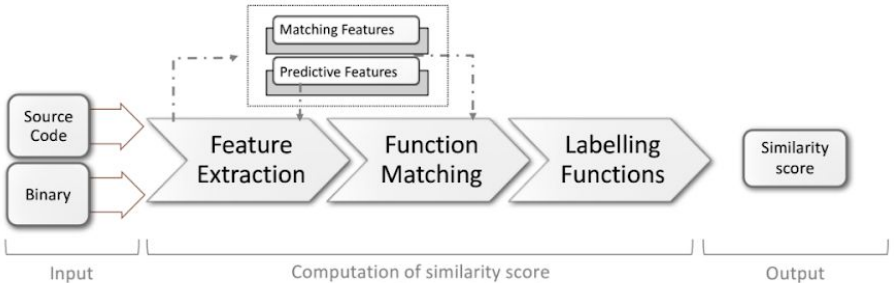
The program uses shadow memory for the observed process, as you can see above. As the program runs within TEMU's secure emulated environment, an execution monitor instruments each memory write to the shadow memory space. In addition, it queries the memory to check if the memory to be written to is clean, meaning it is empty space without a value written to it, or dirty, meaning it contains a value already. If the memory region is dirty, it can determine that a new instruction has been generated. When it has been determined that new instructions have been written, the program is able to find the hidden code and extract it along with its original entry point (OEP) (Kang, 5). Additionally, Renovo is able to detect multiple hidden levels by cleaning dirty memory slates and repeating the process shown above. Since determining whether or not another layer is present is difficult, a timeout is implemented to detect the extraction of newly generated code. If hidden code is not observed as being executed within this time limit, the program is terminated.

While this method of binary filtering is effective, it is not without its vulnerabilities. Many examples of malware that utilize the method of packing are developed with the ability to detect and circumvent emulated environments. Another potential vulnerability is the exploitation of the timeout feature used. If a malicious program were to remain inactive, or stay on a busy loop it could potentially evade detection. A possible solution could be to count the number of instructions executed instead, giving a more accurate metric. Next, we will look at an automated binary filtering and analysis tool related to the legality of code distribution and code plagiarism.

Binary Filtering and Analysis Tools (BinPro - Code Similarity)

The subject of determining similarity between compiled binary files and source code has become increasingly important in recent years with our society becoming evermore computer reliant. When an application is developed and distributed it is required by law that the source code is made publicly available for full transparency and to ensure that there are no hidden, undesirable functions embedded within any code intended for wide release. When these laws, also known as the GNU General Public

License laws, are suspected to be broken, the distributed software is often compared to the publicly available source code to ensure that there has been full transparency. The process of determining this similarity is also important when determining authorship and encompasses processes of reverse software engineering. This analysis can be performed by a popular binary filtering tool known as BinPro, which can analyze and determine the similarity between a given source code and binary file with a high degree of accuracy. It uses statically extracted code features instead of individual instructions, making it a generally static form of analysis, and allows it to scale well with large scale, real world applications.



The diagram above generally describes the process of analyzing both the source code and the binary file. During the first step, feature extraction, BinPro statically pulls features such as string and integer constants, library calls and a function call graph to be compared in the next step. Additionally, the program uses predictive features trained by the programs AI to accommodate for inline functions during compilation (Miyani, 4). The features are extracted and input into a feature table to be matched as well as classified by weight according to how indicative of similarity a match would be.

Next the extracted feature tables are compared using a bipartite matching algorithm to iteratively match and weigh the compared/matched function nodes in the function call graph. The weights are calculated based on both the inherent value of how indicative of a match the features are (i.e. a matched string constant is more indicative than a mathed library call), as well as the degree of similarity that is found between the source code and binary files.

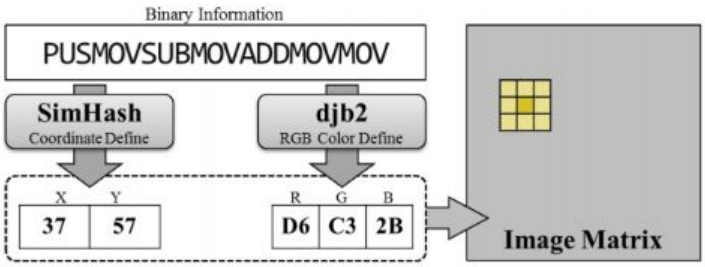
While the scores are important to the matching process, what's more valuable is the labeling of matching features in the next step of the process and the percentage of which are labeled “matched”. The program traverses every edge in the bipartite graph and labels each pair as “matched”, “unmatched” or “multi matched” based on the found number of relations for a given feature. One-to-one matches are the most indicative, and thus the number of pairs labeled “matched” are the most important for the produced similarity score (Miyani, 6). By statically extracting different aspects of the source code and binary files, BinPro is able to filter through the files and analyze the likelihood that the compiled binary files were produced by the given source code, allowing us to determine authorship, transparency and can give us an advantage in the reversing process.

Binary Filtering and Analysis Tools (Nero- Malware Classification)

Continuing with our automated binary filtering and analysis processes, we look back into malware extraction and analysis. One of the more prevalent issues in cyber defense and security today is the diversification and variety of malware. By re-using code blocks and using automated malware generation tools, attackers can increase their inventory of malicious programs and bypass different security measures. Therefore, the ability to categorize and define these malware variations has become increasingly important in developing cyber defenses. Binary filters are often used during malware analysis and a team of computer science professors at Hanyang University in Seoul, Korea have been working to develop a binary filtering method using visualization to compare and classify families of

malware based on color matrices. This static analysis method operates based on opcode patterns extracted from disassembled binaries and has proven to be 95% accurate on average at classifying malware families (Han, 5).

This method of binary visualization works by disassembling malware binary files and analyzing opcode instruction sequences. The assembly code is parsed through and divided into instruction sequences using a delimiter, only taking the code opcodes (i.e. mov, add, sub, pop).



After extraction, each sequence is given a coordinate and an RGB color code, each produced by its own hash function. The strength of this method is found in the hash algorithms used in determining the location and appearance of each pixel in the produced image. SimHash is a local sensitive hash function used to determine the coordinate of each opcode sequence's pixel. It produces coordinates based on the degree of similarity to other sequences of data, meaning similar opcode sequences will have similar locations on the final image. Djb2 is the function used for determining RGB codes by defining 8-bit values calculated for each red, green and blue color (Han, 4). This binary filtering method produces a matrix of pixels which, when compared to other images produced using the same method, analyzes the similarity to a high degree of accuracy.

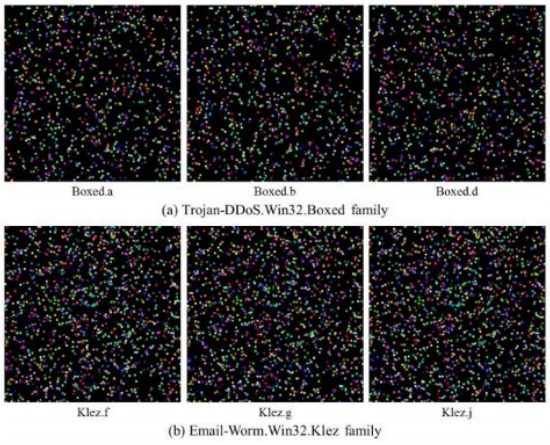


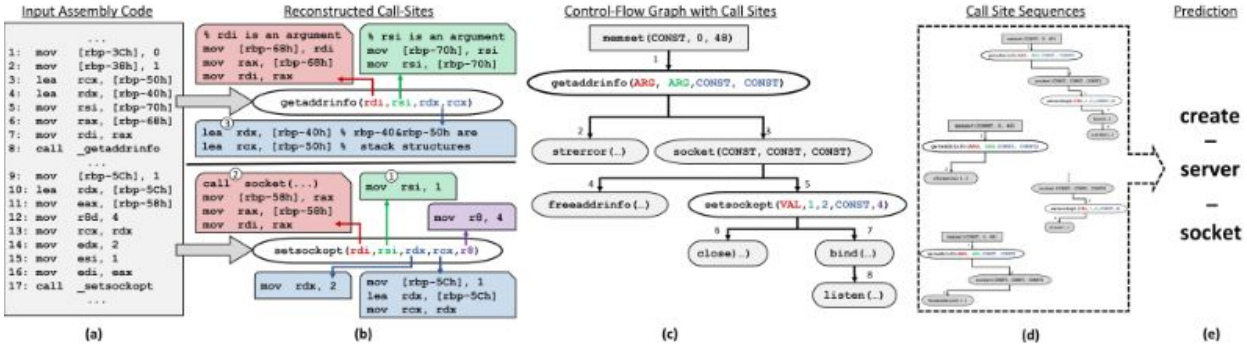
Table 2. Similarities between image matrices of malware

		Boxed family			Klez family			Evol family		
		.a	.b	.d	.f	.g	.j	.a	.b	.c
Boxed family	.a	1	0.968	0.936	0.450	0.449	0.443	0.266	0.267	0.267
	.b	0.968	1	0.932	0.449	0.449	0.442	0.266	0.266	0.267
	.d	0.936	0.932	1	0.448	0.448	0.442	0.265	0.266	0.266
Klez family	.f	0.450	0.449	0.448	1	0.999	0.965	0.263	0.263	0.266
	.g	0.449	0.449	0.448	0.999	1	0.965	0.263	0.263	0.264
	.j	0.443	0.442	0.442	0.965	0.965	1	0.263	0.263	0.264
Evol family	.a	0.266	0.266	0.265	0.263	0.263	0.263	1	0.986	0.960
	.b	0.267	0.266	0.266	0.263	0.263	0.263	0.986	1	0.974
	.c	0.267	0.267	0.266	0.264	0.264	0.264	0.960	0.974	1

Above you can see examples of visualizations produced from malware variations using this method and the degree of similarity between different families and variations. As you can see, this method is highly effective at classifying which family a given malicious binary may be a part of, which gives us a better idea of how to process and manage them. The only downside to this method is that it requires a collection of binaries to compare to and can not classify specific malicious blocks of code on its own. However, with the proper resources, this method of binary filtering proves to be an effective tool in the process of analyzing malicious binaries.We will now take a look at an automated binary filtering and analysis process involving stripped binaries.

A major challenge in the world of binary reverse engineering is managing and effectively analyzing stripped binary executables with a low amount of syntactic information and that contain no debug information. Many of the commonly used models are used to analyze high level languages that are syntactically rich, such as C++, Java and Python, however they are rarely applicable to more unique challenges in reversing binaries, such as analyzing stripped binaries. Since these low level executables often give engineers little to work with, it can be tedious and difficult to analyze and decode these files without the help of an additional tool, such as a trained neural network. A team of researchers at Technion Institute of Technology in Haifa, Israel have developed a platform for analyzing and filtering such files, a model they call "Nero". This platform has been trained and tested to analyze a stripped executable file and predict a program's target name and functionality using machine learning.

Nero works using static analysis to locate and reconstruct external API calls and determine their argument types in the process. These calls are then inserted into a control flow graph created by the platform to be used in the analysis of the function call order, which is essential to Nero's operation. The trained neural network then learns all of the potential function call paths, after which the program's target name is generated by Nero's decoder. Below you can see Nero's steps of analysis, which will be discussed in greater detail below.



To reconstruct the call sites, each external call instruction is analyzed to determine the number of arguments passed, as well as their data types. Information for each argument is gathered, where this information is then associated with each argument and connected to the function call. Having the history of each argument passed serves one of two potential purposes: determining the values passed and determining the type of value passed. When constructing the CFG, function arguments are either labeled argument (ARG), locally created value (VAL), global value (GLOBAL), or unknown constant value (CONST). When an argument's value can be determined by analyzing the function's call sites, it is labeled as its value (David, 3) . For example, as you can see above in figure (b), the values in the rsi and r8 arguments of the function call "setsockopt()" are initialized to 1 and 8 respectively, and in the CFG, figure (c), they're represented as their value.

Once the control flow graph has been extracted, the model determines every possible function call path, as can be seen above in the figure (d). In the assembly code of the stripped binary, each function appears in a random order that has no indicative value, however by analyzing the possible paths we can get an idea of the function of the program. To produce the final target name, the file is analyzed by Nero's AI, represented as a set of potential call set sequences. It follows the general encoder-decoder paradigm with attention for sequence-to-sequence models, modified to where the input is a set of sequences, as opposed to a single sequence (David, 4). The AI, trained using a vocabulary of function call names, as well as their arguments, tokenizes the call sequence set and analyzes each possible path. It then

determines the most indicative, or valuable path for determining the target name, which is yielded after analyzing the strongest path.

Conclusion:

From the tools and processes we have surveyed, none seem to embody the idea of, “one tool serves all needs.” While some tools seem to encompass a larger variety of features than others, it is difficult to judge the overall superiority of each binary filtering and analysis tool we surveyed, given that they usually intend to solve very different issues in regard to the field. Static binary filtering and analysis processes will continue to be developed to solve the theoretical problems of the field, but will always have scalability and implementation problems when faced with practical issues. Binary filtering and analysis tools, whether static, or dynamic in nature, will most likely solve scalability issues, despite requiring larger efforts to actually implement. These tools will also seem to vary quite drastically in regard to insight, generality, and precision of the application. In spite of all of these differences, these tools and processes will continue to comprise and support the first few necessary steps to extract pivotal information from a binary, and commence the overall reverse engineering process.

Works Cited

Hanov, Steve. "Static analysis of binary executables." *University of Waterloo* (2009).

Kruegel, Christopher, et al. "Static disassembly of obfuscated binaries." *USENIX security Symposium*. Vol. 13. 2004.

Xu, Zhichen, Barton P. Miller, and Thomas Reps. "Safety checking of machine code." *ACM SIGPLAN Notices* 35.5 (2000): 70-82.

Conti, Gregory, and Erik Dean. "Visual forensic analysis and reverse engineering of binary data." *Black Hat USA* (2008).

Bardin, Sébastien, et al. "The BINCOA framework for binary code analysis." *International Conference on Computer Aided Verification*. Springer, Berlin, Heidelberg, 2011

Bordeaux, Lucas, Youssef Hamadi, and Claude-Guy Quimper. *The bit-vector constraint*. Technical Report 86, Microsoft Research, 2006.

Cyphert, John, et al. "Refinement of path expressions for static analysis." *Proceedings of the ACM on Programming Languages* 3.POPL (2019): 1-29.

Song, Dawn, et al. "BitBlaze: A new approach to computer security via binary analysis." *International Conference on Information Systems Security*. Springer, Berlin, Heidelberg, 2008.

Kang, Min Gyung, Pongsin Poosankam, and Heng Yin. "Renovo: A hidden code extractor for packed executables." *Proceedings of the 2007 ACM workshop on Recurring malcode*. 2007.

Miyani, Dhaval, Zhen Huang, and David Lie. "Binpro: A tool for binary source code provenance." *arXiv preprint arXiv:1711.00830* (2017).

Han, KyoungSoo & Lim, Jae & Im, Eul Gyu. (2013). Malware analysis method using visualization of binary files. Proceedings of the 2013 Research in Adaptive and Convergent Systems, RACS 2013. 317-321. 10.1145/2513228.2513294.

David, Yaniv, Uri Alon, and Eran Yahav. "Neural reverse engineering of stripped binaries." *arXiv preprint arXiv:1902.09122* (2019).