On Adding Context to Automated: NET Malware Analysis

Anushka Virgaonkar and Chaitanya Rahalkar
Georgia Institute of Technology, Atlanta, USA

avirgaonkar3@gatech.edu
cr@gatech.edu

Abstract: Malware analysis benefits substantially with the help of automation. When it comes to analysing .NET malware samples, there is a dearth of automated analysis tools that provide quality results. Streamlining the malware analysis workflow to assist in completing the process in a timely manner is another challenging task. We determine that adding context to each piece of extractable information could help an analyst in understanding the functionality of the .NET sample better. In this paper, we introduce a standalone command-line application developed in Python, designed to assist analysts in .NET malware analysis. We follow a static analysis approach to extract features from the samples, to identify higher-level capabilities and to provide exact indicators of compromise. We do not rely on dynamic analysis as it only follows one path of execution. We compare the results of the tool with similar existing tools that can analyse .NET samples. Through a qualitative evaluation, we showcase the utility of the tool in terms of providing significant insights to a malware analyst. We study openly published Malware Analysis Reports (MARs) that are generated through extensive analysis and observe how the tool can provide the same insights in a simple and reliable manner.

Keywords: Malware analysis, .NET, Reverse engineering, .Net malware

1. Introduction

An exponential growth in malware written in .NET compliant languages is being observed over the past decades.

Malware analysts do not receive much support when it comes to analysing .NET samples. The common tools and techniques that are used for analysing C/C++ malware do not work on .NET samples. For instance, .NET samples do not list the imports and exports in the PE sections, unlike C/C++ samples. The .NET samples only import the _*CorExeMain* function from mscoree.dll instead. The IDAPro disassembler is not useful for .NET samples. The open-source disassembler *dnSpy* is used instead. *dnSpy* currently does not have support for plugins like IDAPro does. This makes integrating handy debugging scripts, which are used frequently by malware analysts for a myriad of analysis processes, an infeasible task. Most of the .NET samples are obfuscated. The open-source de-obfuscator and unpacker de4dot is utilized. The rise in .NET malware, the lack of skilled reverse engineers who understand .NET samples and the lack of highly supportive .NET malware analysis tools make the analysis process challenging.

Few tools have been developed in recent years with the purpose of streamlining the analysis process. Their goal ranges from detecting capabilities in the samples to extracting configurations automatically. However, they lack in providing context for each piece of information they extract. After running the sample through the tool, obtaining the results, and studying the results, the malware analysts would have to face the arduous task of inspecting the disassembly or decompiled code of the sample for getting a clearer idea about the functionality of the sample (Trizna, D., Demetrio (2023)). As a result, the malware analysis process becomes time-consuming and challenging.

For automating .NET malware analysis, a requirement for a tool that does most of the heavy lifting for a malware analyst in terms of providing significant insights is recognized. For providing such insights, the tool could add more context to each piece of information it presents. The added context would assist in uncovering significant bits of information from the sample and piece them together to reveal the story about the malware. The malware analyst would benefit from the context and would be able to reverse engineer and understand the malware functionality in its entirety. Context comprises of Features and Sub-features. Features are the Classes, Namespaces, Strings, Methods and API calls present in the samples. Sub-features provide relevant information about the extracted Features. For instance, the parameters passed to an API call would be the Sub-features of the API Feature.

With the addition of context to every piece of information being extracted by the tool through automated analysis, we aim to achieve 3 main goals:

- To provide the malware analyst with the higher-level capabilities present in a sample.
- To find the exact indicators of compromise and artifacts from the sample.
- To design this tool such that it is able to do most of the heavy lifting for malware analysts.

2. Background Research

Many tools that perform static analysis on binaries have been developed. Most of them predict malicious behaviour by capturing the API calls in the binary (Noman et al. (2021), Saini et al. (2014)). However, they can give false positives because API calls used for malicious functionalities are also found in benign samples. Additionally, malware authors often insert dummy API calls to derail analysis tools (Kunwar and Sharma (2016)). To create a more accurate system, we need to combine various static analysis methods and consider dynamic analysis integration (Lee et al. (2011)).

The concept of context involves including reliably detectable information corresponding to the Features in the sample. Features include API calls, Classes, Namespaces, Functions, and Strings present in the binary. Additional information such as function length, references, count, and parameters can enhance analysis. Bandodkar et al. (2024) found that function length can significantly aid malware classification when combined with other Features, offering a fast and scalable classification method. By providing Sub-features for each extracted Feature, we add information that helps determine functionality accurately. Control flow analysis has also proven valuable in automated malware analysis (Shan et al. (2019)).

While previous research efforts used machine learning algorithms for Sub-feature analysis, our study avoids this approach due to the complexity in obtaining reliable malware datasets and the inconsistency of machine learning results. Instead, we focus on concrete data extracted directly from PE files and MSIL architecture. .NET's additional metadata tables provide more Feature information than x86/x64 binaries, allowing our tool to extract context that would be impossible to obtain from traditional binaries.

3. Existing Solutions

capa is an open-source tool used for detecting the capabilities present in a sample with the help of a rule set. For instance, if a sample is making an HTTP request, creating a process or logging keystrokes, *capa* can identify such functionalities. *capa* makes use of the open-source tools *dnfile*, for parsing the .NET metadata and *dncil*, for extracting the instructions in the sample, to emit Features such as the API calls, classes, Namespaces, numbers and strings used in the program. *capa* attempts to find rule matches in the sample. A rule corresponds to a specific functionality and contains a list of the Features necessary for its implementation.

The latest version of *capa* was developed with the goal of processing .NET samples. It performs analysis on .NET samples and provides rule matches that signify the presence of malicious capabilities. But it does not provide enough context for each capability. The capabilities extracted can be said to be more high-level. Moreover, the tool was not designed with the purpose of extracting host-based indicators and network-based indicators. Any special conditions that may be of interest to a malware author are not detected either. For instance, if a sample attempts to start a process, *capa* would show a rule in its output that indicates that *capa* has detected the creation of a new process. Although, it does not provide additional information of the process, such as the name of process, or the conditions under which the process has been created.

It is also not detected whether a functionality would actually be executed by the malware or if it was merely implemented in the program. Furthermore, after studying the rule matches obtained from *capa*, a malware analyst would still have to manually look at the disassembly to find the context and complete the analysis. In complex .NET samples such as the *DARKCRYSTALRAT* variant malware, which gives 64 *capa* rule matches, it is a tedious process to manually identify the context for each rule match.

Automated malware configuration extraction is another process used for finding the indicators-of-compromise (IOCs). For each malware family, a script is written to obtain the configuration data such as the IP addresses of the C2 servers, names of the files dropped by the sample, values of the registry keys created and so on. The malware family of the sample is found by identifying the script that is able to successfully extract the configuration information (Maniriho et al. (2022)). By learning more about the malware family, the analyst can efficiently analyze the given sample (Manna et al. (2022)).

However, this requires maintaining a list of scripts corresponding to each malware family, which would be needed to be kept up to date. If a script for a malware family is not developed, then the sample belonging to this family would go undetected. Furthermore, malware authors could introduce variations to prevent successful configuration extraction (Afianian et al. (2019)).

4. Methodology

Our approach utilizes static analysis techniques enhanced with emulation to detect functionality within .NET samples. This method allows us to simulate execution paths and analyse data flow without actual execution.

The use of dynamic analysis is not considered because it only allows us to view only one execution trace (Muralidharan et al. (2022)). Since we aim to uncover all the paths of execution taken by the malware, relying on static analysis is a suitable option. Static analysis does not allow us to view the exact values of the variables, or data stored in registers or in memory, unlike dynamic analysis. To track data flow and simulate runtime behaviour, we implement a stack-based emulator within our static analysis framework. This emulator processes intermediate language (IL) instructions, maintaining a simulated stack to observe data manipulation and control flow, thereby inferring runtime-like behaviour without executing the sample. Our tool employs an advanced static analysis approach that incorporates a stack-based emulator to simulate certain runtime behaviors without actual execution. This method enables the analysis of multiple execution paths, maintaining data flow, and deriving runtime-like behavior within the confines of static analysis. By emulating key aspects of execution, the tool provides insights typically associated with dynamic analysis while preserving the comprehensive coverage inherent to static methods.

By emulating IL instructions, our tool maintains data structures that simulate the program's stack and memory. This enables us to observe data flow and determine the values of variables and parameters at different points in the program, facilitating a deeper understanding of the sample's functionality through static means. Through this enhanced static analysis approach, we identify important artifacts, including strings and indicators of compromise, which are instrumental in generating signatures for malicious samples. This method ensures comprehensive analysis without the need for dynamic execution. The first step would be to extract and emit the Features present in the binary. .NET samples come with metadata tables that hold information about classes, Namespaces, methods, properties and so on. The following tables are useful in extracting, emitting and finding the correlations between the Features and Sub-features.

The tool operates as a standalone command-line application, allowing users to input .NET binaries for analysis. Developed in Python, it provides a straightforward interface where analysts can specify the target sample and receive detailed reports on extracted features, identified capabilities, and potential indicators of compromise. The tool's accessibility and ease of use aim to enhance the malware analysis workflow for security professionals." .NET samples come with metadata tables that hold information about classes, Namespaces, methods, properties and so on. The following tables are useful in extracting, emitting and finding the correlations between the Features and Sub-features.

4.1.1 TypeDef table

The *TypeDef* table holds information about the classes (or interfaces) present in the .NET binary. Each row represents a class (or interface) and holds references to the methods and fields belonging to that class (or interface).

The *TypeDef* table has the following columns:

- *Flags* (a 4-byte bitmask of type *TypeAttributes*)
- *TypeName* (an index into the String heap)
- *TypeNamespace* (an index into the String heap)
- *Extends* (an index into the *TypeDef*, *TypeRef*, or *TypeSpec* table)
- *FieldList* (an index into the Field table)
- *MethodList* (an index into the *MethodDef* table)

The information about inheritance hierarchies is also stored in this table. The *Extends* column of the *TypeDef* table gives the information about the base class that is extended by this (derived) class. The *InterfaceImpl* table provides information about the interfaces that a class implements.

4.1.2 TypeRef table

The *TypeRef* table has the following columns:

- *ResolutionScope* (an index into a *Module, ModuleRef, AssemblyRef or TypeRef* table, or null)
- *TypeName* (an index into the String heap)
- *TypeNamespace* (an index into the String heap)

4.1.3 MethodDef table

The *MethodDef* table holds information about the methods in the sample. Each row represents a method. The methods in the *MethodDef* table belong to certain classes. Each row in the *MethodDef* table always has one, and only one, owner row in the *TypeDef* table.

The *MethodDef* table has the following columns:

- RVA (a 4-byte constant)
- *ImplFlags* (a 2-byte bitmask of type *MethodImplAttributes*)
- Flags (a 2-byte bitmask of type *MethodAttributes)*
- Name (an index into the String heap)
- Signature (an index into the Blob heap)
- *ParamList* (an index into the *Param* table)

4.1.4 MemberRef table

The *MemberRef* table provides us with the combination of two types of references. one is to Methods and the other is to the Fields of a class. They are known as *MethodRef* and *FieldRef*, respectively. An entry is made into the *MemberRef* table whenever a reference is made in the CIL code to a method or field which is defined in another module or assembly. The *MemberRef* table has three columns:

- Class (it stores an index into the *MethodDef*, *ModuleRef*, *TypeDef*, *TypeRef*, or *TypeSpec* tables)
- Name (an index into the String heap)
- Signature (an index into the Blob heap)

4.1.5 MethodSemantics Table

The MethodSemantics table provides an association between the methods and the properties that they retrieve or modify. The MethodSemantics table has the following columns:

- Semantics (a 2-byte bitmask of type MethodSemanticsAttributes)
- Method (an index into the MethodDef table)
- Association (an index into the Event or Property table)

4.1.6 Property table

The Property table has the following columns:

- Flags (a 2-byte bitmask of type PropertyAttributes)
- Name (an index into the String heap)
- Type (an index into the Blob heap)

4.1.7 PropertyMap table

The *PropertyMap* table has the following columns:

- Parent (an index into the *TypeDef* table)
- PropertyList (an index into the Property table)

4.1.8 Namespace Feature

The Namespace Feature provides information about the namespaces defined in the .NET sample. These namespaces can be user-defined or standard, imported .NET namespaces. We can find the Namespace of the class in the *TypeDef* column and emit the Namespace Feature accordingly.

4.1.9 Class Feature

The Class Feature provides information about the classes defined in the .NET sample. These classes can be user-defined or standard, imported .NET classes. The *TypeDef* table holds the list of classes and other related information about the classes in its rows. We can find the Namespace of the class in the *TypeDef* column and emit the Namespace Feature accordingly. The Sub-features of the Class Feature can be extracted using the *TypeRef* table. When a Class feature is emitted, the Namespace feature is emitted as well.

4.1.10 Method Feature

The Method Feature provides information about the methods defined in classes in the .NET sample. Methods belong to classes. The information about the relationship between methods and classes can be found in the *MethodRef* table. The *subFeature* Field can also be extracted using this table. The *MethodDef* table holds the list of all methods present in the .NET sample. We parse this table to obtain information about the methods such as their name, list of parameters, signatures and so on.

4.1.11 API Feature

The API Feature is the subset of methods that are part of the standard .NET APIs. When writing rules, these API Features would be included as they are the standard way for implementing different functionalities. When emitting a Method Feature, the corresponding Class and Namespace Feature is emitted also.

4.1.12 Property Feature

The Property Feature holds information about the properties belonging to classes in the .NET sample. The properties are fields that are modified using methods. These are getter and setter methods that are being with the get_ and set_ prefix. The extraction of the Property Features involves parsing three tables and associating the extracted information. The *MethodSemantics* table links methods from the *MethodDef* table and Properties in the Property table. The *PropertyMap* table links the properties in the Property table and the classes in the *TypeDef* table. By finding these associations, we can find which method modifies or retrieves a property and which class (and) Namespace does this property belong to.

Malware authors use properties to set certain values while implementing a functionality. These are usually properties belonging to standard .NET classes that have been imported in the .NET sample. For instance, *WindowStyle* property of the System.Diagnostics Namespace and *ProcessStartInfo* class is used to get or set the window state to use when the process is started. The *MachineName* property of the System Namespace and the Environment class is used to obtain the NetBIOS name of the local computer. By including properties in writing rules, we can uncover the functionality implemented by the malware author and understand the capabilities present in the sample better.

The *ldfld*, *ldflda*, *ldsfld*, *ldsflda*, *stfld* and *stsfld* opcodes are also used to retrieve or modify a property value. The opcodes in the instructions of the .NET sample are search to detect whether they stored or load a property. Then the operand is parsed to obtain information about the property and the Property Feature in emitted. While emitting the Property Feature, we also emit the Class and Namespace Feature.

4.1.13 String Feature

The open-source library *dncil* is used to extract the String Feature. It emits String tokens. We parse these tokens to find the information about the strings present in the .NET sample and emit them as String Features.

The *ldstr* instruction pushes a new object reference to a string literal stored in the metadata. The operand of this instruction points to a string. We search the instruction present in the .NET sample to find the instruction having the *ldstr* opcode and then we retrieve the operand to find the string. We then emit this String as a Feature.

String Features can often provide the indicators of compromise. For instance, an IP address can be stored as a string, or a malware may create a mutex which is stored as a string to avoid re-infecting the victim's machine. We can include the String Feature in rules to detect certain capabilities. For instance, a .NET sample may employ an evasion technique where it searches for strings belonging to analysis tools. We can write a rule where we include those strings. The strings can be ollydbg.exe, ProcessHacker.exe, tcpview.exe, procmon.exe, regmon.exe, procexp.exe and so on. Furthermore, we can implement the use of regular expressions for matching patterns rather than exact values of the strings.

4.1.14 Number Feature

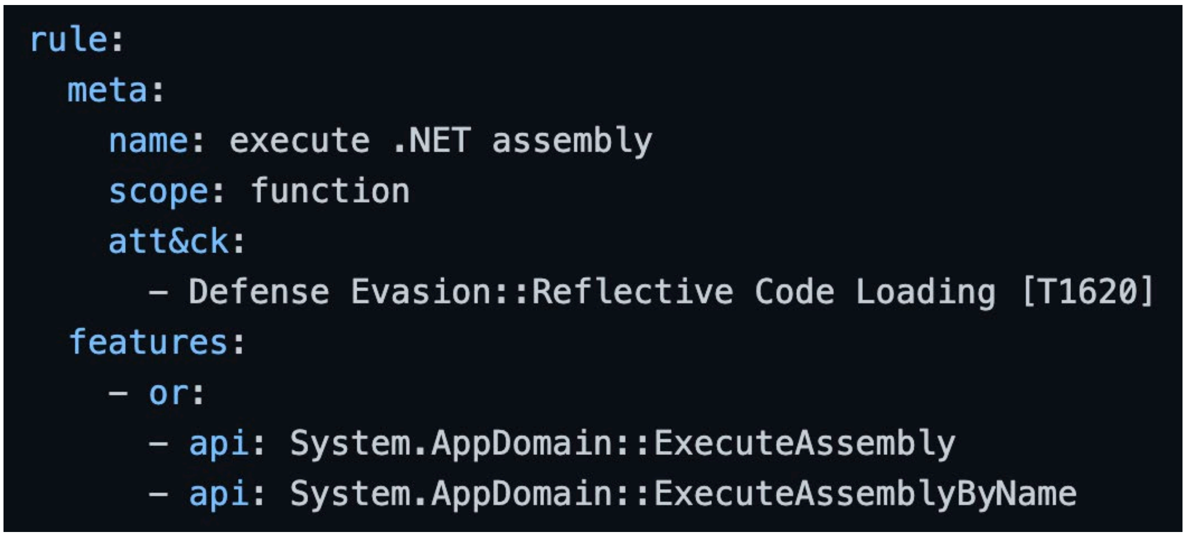
The numbers present in the .NET sample can be emitted as the Number Feature. The opcodes beginning with *ldc* are used to load integers and floating numbers. We search the instruction present in the .NET sample to find the instruction having the *ldc* opcode and then we retrieve the operand to find the number. We then emit this Number as a Feature.

Malware authors use number for implementing certain functionalities. For instance, for the checking the OS version, first, the API *GetVersion* may be called and then the result of the API call is compared with the number 5 for Windows 2000, 10 for Windows 10, 1 for Windows XP or 6 for Windows Vista.

4.1.15 Identifying capabilities

Capturing Features is important because they help in identifying higher-level capabilities. For instance, the Process.Start API call is used to create a new process. If we emit this API call Feature, we can write the logic to infer that the .NET sample creates a new process. We can also find the exact name of the created process once we find the parameters passed to the call.

After extracting Features and Sub-features, we would need to develop and maintain a rule set. Rules follow a YAML-like syntax and they signify the presence of a particular capability. The following rule is used to detect whether the .NET sample executes a .net assembly in memory. This is a very common functionality observed in malware samples, where some code is downloaded from the internet and it runs in memory without ever being stored on disk so as to not leave any evidence. If you look closely, the rule would be matched if either the API call ExecuteAssembly or ExecuteAssemblyByName is present in the .NET sample.



```
rule:
  meta:
    name: execute .NET assembly
    scope: function
    att&ck:
      - Defense Evasion::Reflective Code Loading [T1620]
  features:
    - or:
      - api: System.AppDomain::ExecuteAssembly
      - api: System.AppDomain::ExecuteAssemblyByName
```

Figure 1: Execute .NET assembly rule

4.2 Identifying Indicators of Compromise

The next and most important step is to identify the indicators of compromise (IOCs) and other important artifacts within a .NET sample. While dynamic analysis is commonly used to obtain IOCs, it is limited by its single-path execution, which often fails to capture all potential execution paths and the corresponding artifacts. To address these limitations, we adopt a static approach that emulates certain runtime behaviors without directly executing the sample.

4.2.1 Stack-Based instruction approach

Our approach involves implementing a stack-based emulator to replicate the data flow through the sample's code. This emulator processes the intermediate language (IL) instructions found in .NET binaries, extracting the values of variables, registers, and memory locations. By maintaining a stack data structure, the emulator simulates how data is pushed, popped, and manipulated during execution.

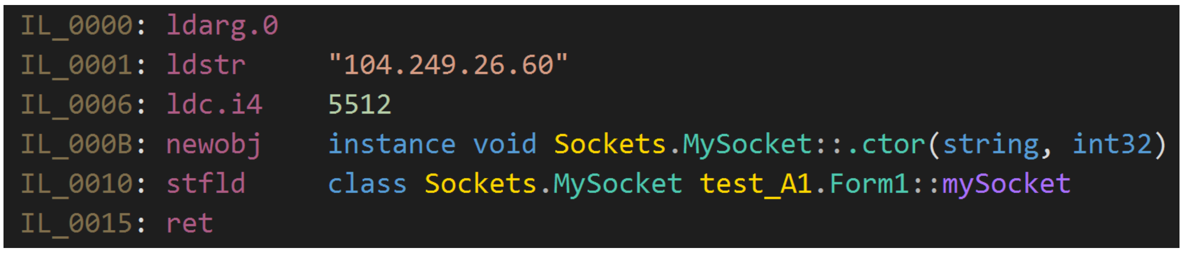
Instruction decoding involves processing opcodes like *ldstr*, which pushes string literals onto the stack, and *ldc.i4*, which pushes 32-bit integers. These values are decoded and stored on the stack for further use. During object creation, the *newobj* opcode pops arguments such as IP addresses or port numbers from the stack and passes them to a constructor to instantiate a new object. A reference to the newly created object is then pushed back onto the stack. For method calls, opcodes like *callvirt* are used to invoke methods, utilizing arguments popped from the stack and generating return values, which are subsequently pushed back onto the stack for further processing.

This process mimics the runtime behaviour of the code and enables the identification of IOCs without actual execution. For example, in the MSIL instructions for the user-defined method Form1.Form1_Load, the stack emulator can trace the initialization of a *Sockets.MySocket* object and extract its arguments (e.g., an IP address and port). These arguments can then be correlated with subsequent method calls, such as a *TcpClient.Connect* API call, to infer specific IOCs like connection endpoints.

This approach offers several advantages. It enables path exploration by evaluating multiple code paths in a controlled manner, thereby overcoming the single-path limitation of dynamic analysis. Through static runtime emulation, the methodology bridges the gap between purely static and fully dynamic analysis by simulating dynamic aspects without actual execution. Additionally, the stack-based logic enhances scalability, allowing the system to automate processes and efficiently handle multiple samples. By providing a static yet dynamic-like perspective of code execution, this emulation offers valuable insights into malware functionality. The reliance on a small set of critical opcodes ensures the emulator remains lightweight while effectively extracting meaningful artifacts.

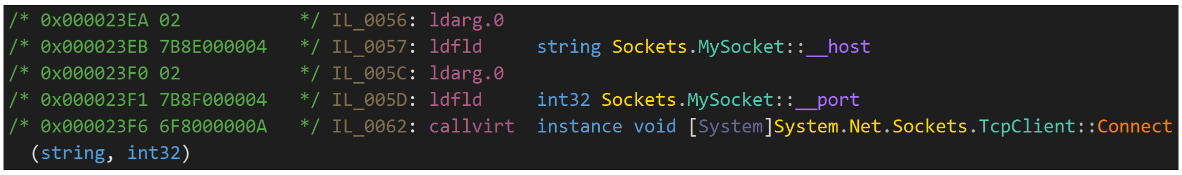
Here is an example for depicting the meaning of a call scope and understanding the process of identification of indicators of compromise using a stack-based instruction emulator:

The following code contains the MSIL instructions present in the user-defined method Form1.Form1_Load, that are used to instantiate the *ctor* method of the user-defined class *Sockets.MySocket*. The method takes two arguments: IP address and port.



```
IL_0000: ldarg.0
IL_0001: ldstr      "104.249.26.60"
IL_0006: ldc.i4     5512
IL_000B: newobj     instance void Sockets.MySocket::.ctor(string, int32)
IL_0010: stfld      class Sockets.MySocket test_A1.Form1::mySocket
IL_0015: ret
```

Figure 2: Instructions in *Form1.Form1_Load*



```
/* 0x000023EA 02           */ IL_0056: ldarg.0
/* 0x000023EB 7B8E000004   */ IL_0057: ldfld     string Sockets.MySocket::__host
/* 0x000023F0 02           */ IL_005C: ldarg.0
/* 0x000023F1 7B8F000004   */ IL_005D: ldfld     int32 Sockets.MySocket::__port
/* 0x000023F6 6F8000000A   */ IL_0062: callvirt  instance void [System]System.Net.Sockets.TcpClient::Connect
  (string, int32)
```

Figure 3: Instructions in *Sockets.MySocket*

The method *Sockets.MySocket*(string host, int port) attempts to connect to the IP using the imported .NET API *System.Net.Sockets.TcpClient::Connect().* Current static analysis tools can detect this standard API call and determine that the .NET sample attempts to use socket communication. However, they would not present the exact IP address and port number.

To extract the IP address and port number, or more generically, to obtain the parameters values passed to an API call, we would need to make use of control flow techniques to keep track of memory values. We could make use of the instructions in the relevant sections of the code. The opcodes and operands are obtained from *dncil*. In figure 1, The *ldstr* opcode pushes a new object reference to a string literal stored in the metadata on the stack. The *ldc.i4* opcode pushes a supplied value of type int32 onto the evaluation stack. At IL_000B a newobj opcode is used to create a new object *Socket.MySocket*, pushing an object reference onto the evaluation stack. The arguments (host and port) are popped from the stack and passed to *ctor* for object creation. A reference to the new object is pushed onto the stack. We implement a stack data structure and push the arguments 104.249.26.60 and 5512 on it. We hard-code the logic for each relevant opcode. This task may seem tedious but the list of relevant opcodes is minimal. By understanding the behavior of *newobj*, we can determine precisely the values of the arguments passed to the constructor.

Now, in *Sockets.MySocket* method (Figure 4), the argument *__host* would be the IP address and *__port* would be the port number. The *ldfld* opcode is being used to push the value of a field in a specified object onto the stack. So, the values of __host and __port would be stored on the stack. The *callvirt* opcode is used to call the specified method in the operand and the method arguments are popped from the stack. The method call is performed with these arguments. When complete, a return value is generated by the callee method and sent to the caller. In this manner we can find that the arguments passed to the standard API *System.Net.Sockets.TcpClient::Connect* were the IP address 104.249.26.60 and the port number 5512.

With the first step of our processing, we are able to extract the important Features such as APIs, classes, Namespaces, strings, numbers for .NET samples. With the help of *dncil*, we can access function handlers and parsing instructions. We implement our emulator to maintain the stack and replicate the execution without actually running the .NET sample. We observe that samples may have higher depths in their method calling procedures. We have implemented our tracing algorithm for depths of levels up to 4. We do not exceed this depth to avoid the problem of path explosion.

5. Evaluation

For quantitatively evaluating our tool, I ran it on 32 .NET samples and compared the output with the output generated by other tools. Here are the results obtained:

- The tool identified an average of 33.7% more capabilities per sample compared to capa. This improvement is attributed to our comprehensive rule set tailored for .NET sample processing.
- When benchmarked against malconf, our tool extracted 41.5% more configuration data on average. Unlike malconf, which depends on predefined scripts for each malware family and may miss configurations for unsupported families, our tool's generic approach ensures broader applicability across diverse .NET samples.
- MARs detect 9.4% more capabilities than the tool. We can reduce this number by adding more rules and updating existing ones frequently.
- MARs exceed the API call inferences by an average of 18.5% per .NET sample. The tool lacks in this area due to the problem of path explosion and obfuscation.

Overall, the tool performs better than *capa*, which can be contributed to the fact that the tool relies on a rule set containing 60+ rules for .NET samples.

The tool also performs better than automated malware configuration extractors in terms of the quantity of configuration data it extracts. Moreover, automated malware configuration extractors are not generic in nature; they are not able to process any given .NET sample. Our tool is generic and can process any given .NET sample. Even for the samples for which automated malware configuration extraction scripts were written, the tool provided with, on an average, an 8.4% higher number of configuration data.

MARs are generated with the involvement of a reverse engineer. They would have to include every single detail about the malware's functionality. The success of the tool is not in providing with every single output that the MAR includes, but it is in providing the malware analyst with useful insights quickly and reliably. The malware analyst should be able to perform their analysis task smoothly with the help of the tool. With the three case studies we observed the instances where the malware analyst benefits largely by understanding the output of the tool.

6. Limitations

One of the major limitations of the tool is path explosion. When deciding the depth for defining the call scope, we chose it to be 4. Beyond that, we were experiencing performance issues. Out of the 32 samples analysed, 5 of them gave significantly low results in terms of missing high and medium priority insights, due to path explosion.

Another limitation is code obfuscation. To address this, we have integrated automated de-obfuscation mechanisms into the tool, utilizing de4dot, an open-source .NET deobfuscator and unpacker. This integration enables the tool to automatically de-obfuscate payloads, allowing for accurate detection of artifacts without requiring manual intervention from the malware analyst.

The requirement that the ruleset needs to be kept updated is another limitation. If rules are not kept frequently updated, some higher-level capabilities could go undetected.

Finally, differences were observed between instructions and Features captured by looking at CIL vs through the de-compilation results in *dnSpy*, due to *dncil*.

7. Conclusion

In this paper, we proposed a new tool for automated .NET malware analysis, detailing its design and methodology. The tool demonstrates significant improvements in providing contextual insights, allowing analysts to uncover a higher number of inferences compared to existing tools. Through case studies, we highlighted its potential to streamline analysis workflows by automating complex tasks. While the tool effectively bridges gaps in current methodologies, limitations such as handling obfuscation and frequent rule updates remain. Nonetheless, it serves as a promising supplemental tool that reduces manual effort and provides valuable insights for malware analysis. Future enhancements could further strengthen its impact on analysis workflows.

References

Afianian, A., Niksefat, S., Sadeghiyan, B. and Baptiste, D. (2019) "Malware dynamic analysis evasion techniques: A survey", ACM Computing Surveys, 52(6).

Bandodkar, C., Parashar, U., Rajput, A., Mittal, N., & Khimesara, S. (2024) "Building resilient AI-enabled detection systems for .NET threats: Insights from the DoubleZero wiper," *Nanotechnology Perceptions*, 20(S16)

Borges, A. (2019) ".NET malware threat: Internals and reversing," DEF CON 27 Conference Presentation.

Kunwar, R.S. and Sharma, P. (2016) "Malware analysis: Tools and techniques", In: Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies. New York: Association for Computing Machinery.

Maniriho, P., Mahmood, A. N., & Chowdhury, M. J. M. (2022) "MalDetConv: Automated behaviour-based malware detection framework based on natural language processing and deep learning techniques," arXiv preprint arXiv:2209.03547.

Manna, M., Case, A., Ali-Gombe, A., & Richard III, G. G. (2022) "Memory analysis of .NET and .NET Core applications," Proceedings of the Digital Forensic Research Conference (DFRWS).

Muralidharan, T., Cohen, A., Gerson, N. and Nissim, N. (2022) "File packing from the malware perspective: Techniques, analysis approaches, and directions for enhancements", ACM Computing Surveys, 55(5).

Noman, H.A., Al-Maatouk, Q. and Noman, S.A. (2021) "A static analysis tool for malware detection", In: 2021 International Conference on Data Analytics for Business and Industry (ICDABI), pp.661-665.

Saini, A., Gandotra, E., Bansal, D. and Sofat, S. (2014) "Classification of pe files using static analysis", In: Proceedings of the 7th International Conference on Security of Information and Networks. New York: Association for Computing Machinery, pp.429-433.

Shan, P., Li, Q., Zhang, P. and Gu, Y. (2019) "Malware detection method based on control flow analysis", In: Proceedings of the 2019 7th International Conference on Information Technology: IoT and Smart City. New York: Association for Computing Machinery, pp.158-164.

Trizna, D., Demetrio, L., Biggio, B., & Roli, F. (2023) "Nebula: Self-attention for dynamic malware analysis," arXiv preprint arXiv:2310.10664.