

PEScan: Streamlined Malware Static Analysis

CMP320 2025 Term 2
Scripting Project Assessment
Diego Andrade (2200905)

Abstract

New malware is released constantly, and malware analysts cannot work fast enough to identify, examine, and classify samples. Malware analysis is a lengthy process, with multiple stages where attention to detail is key. Static analysis of a sample gives analysts an overview of the sample and a starting point for their investigation. Thus, automation of tedious static analysis tasks allows researchers to progress through the examination process and can prevent human error. Most malware today comes in the form of portable executable files (PE) for the Windows operating system. These files contain header data that is crucial to gain an initial foothold and beginning to grasp the functionality of a sample. One of the header sections contains the Windows API imports, which indicate the external functions the sample relies on to perform tasks at the operating system level. The PEScan tool aims to automate interpretation and categorization of these API imports in a robust and performant way. Additionally, the tool aims to provide a flexible and effective interface for interacting with other software.

This tool is implemented in the Rust programming language and leverages a variety of libraries to meet these aims; notably, the 'goblin' crate is used for PE header parsing. The tool categorizes the API imports using data scraped from the *MalAPI.io* website. This data is cached using the MessagePack format for maximum performance. The tool employs a range of Rust's design paradigms and data types to maximize performance and ensure reliability. The tool allows for serialization of data in the following formats: TXT, JSON, YAML, TOML, and CSV. Furthermore, a simple and transparent serialization interface can be used to extend serialization functionality and implement more output formats. Data can be output to either STDOUT or to a file, for ease of use with other tools. PEScan has an estimated execution time of 104.4ms and peak memory usage of 4MB. Rust's ownership system ensures that the program is implemented in a completely memory-safe way and error messages contain precise context and a concise backtrace. Multiple build outputs are available: two native 64-bit binaries for Windows and Linux, and a Docker image.

Through benchmarking, boundary testing, and integration testing, it was found that PEScan outperformed a comparable tool. It performed the same task in half the time, consuming two thirds of the memory as the existing tool, whilst providing more granular and adaptable output. This program successfully automated the extraction and categorization of Windows API imports. This means that analysts and researchers can save time and preclude the tedious task of sorting through enormous quantities of APIs that often have similar functions. The tool is limited by redundant fields in data structures that don't affect performance but do affect readability and maintainability. Another limitation is lack of optimization in the plain text output's serialization crate. This introduces potential for improvement by implementing custom serializers to eliminate redundancies.

Table of Contents

1	Introduction	4
1.1	Background	4
1.2	Aims	4
2	Technical Resources.....	5
2.1	Programming Language	5
2.2	Dependencies.....	7
2.3	Implementation	9
2.4	Testing	10
3	Development	11
3.1	Argument Parsing.....	11
3.2	Data Management.....	11
3.3	Output Formatting.....	14
3.4	Main Functionality.....	16
3.5	Benchmarking.....	16
4	Results.....	17
4.1	Robustness.....	17
4.2	Performance	18
4.3	Interoperability.....	20
4.4	Comparison with Existing Tool	21
5	Discussion	22
6	Future Work	23
	References.....	24
	Appendices	25
	Appendix A – Cargo.toml.....	25
	Appendix B – Full Benchmark Code.....	26
	Appendix C – Benchmark Distribution Graphs	28
	Appendix D – Full-Size Memory Usage Graphs	30

1 Introduction

1.1 Background

Cybercrime worldwide has been rising at an alarming rate for multiple years. The annual number of malware attacks worldwide in 2023 was 6.06 billion (SonicWall, 2024), and the estimated cost of malware attacks between 2024 and 2029 is expected to rise by \$6.4 trillion USD (Statista, 2024). This is expected to be a 69.4% growth. Malware analysts cannot keep up with the number of new samples distributed yearly. In 2023 alone, 52,646 unique malware threats were reported by end users of worldwide organizations (Proofpoint, 2024). Of all these threats, the most common format for malware to propagate are EXE files: 56% of malicious file types received on the web globally are EXE files (Check Point Software Technologies, 2024). EXE files are a subset of Portable Executable (PE) files, as PE files can also describe Dynamic Link Libraries (DLLs). A great majority of PE files make use of linked Windows API libraries. This allows programs to perform basic operating-system level tasks. A journal article about Windows API based malware detection concluded that a statistical analysis of Windows API calls can accurately reflect the behaviour of a piece of code based on an analysis framework of over 91% accuracy (Rai and R, 2012, p. 5).

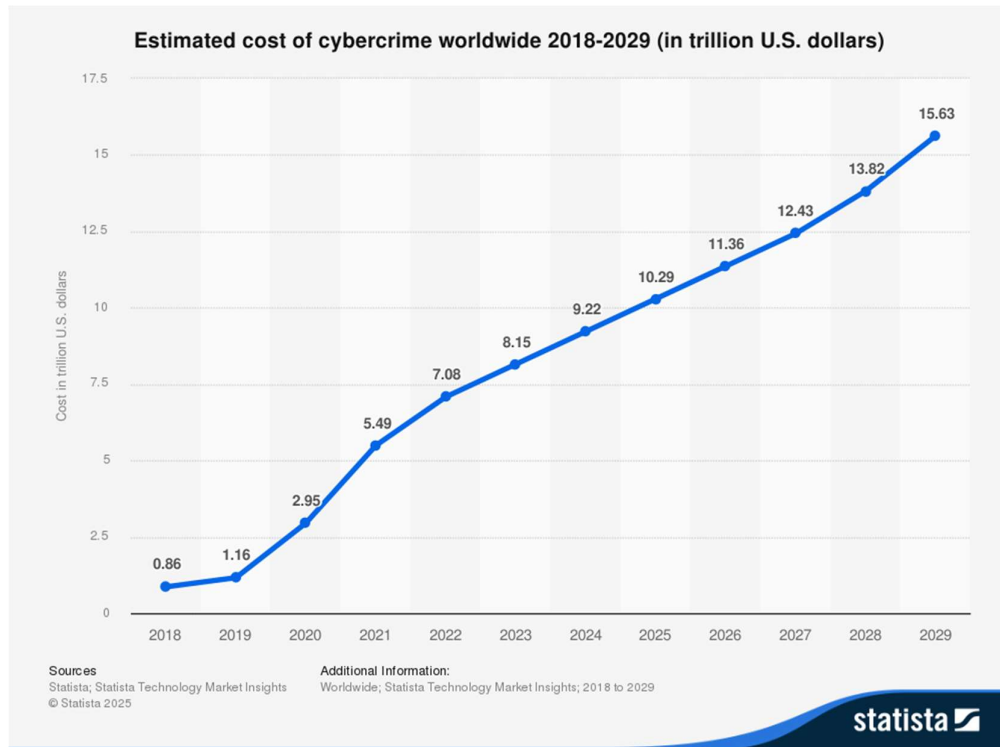


Figure 1: Estimated cost of cybercrime worldwide 2018-2029 (in trillion U.S. dollars)

Static analysis is a technique used to inspect malware samples without executing them, this can be quicker and safer than dynamic analysis (where the sample is analysed during runtime). In a survey on malware analysis techniques by Sihwail, Omar and Zainol Ariffin (2018, p. 4), five papers surveyed used static analysis techniques, three of which based their research on Windows API calls. Two of those papers had an accuracy of over 90% for classifying malware. Investigating Windows APIs has one significant limitation: the sheer volume of different DLLs and API functions used for largely similar tasks; furthermore, malware analysts cannot spare the time to tediously classify each API in the imports header.

1.2 Aims

This report describes a CLI tool that streamlines static analysis by automating classification of Windows API imports. It is meant to be used during a preliminary stage of static analysis to help guide the analyst during decompiled code inspection. The tool was developed with the following aims in mind:

- Robustness
- Performance
- Interoperability

2 Technical Resources

2.1 Programming Language

PEScan was developed using the Rust programming language, it was chosen because the author was familiar with the language and ecosystem for developing CLI tools. Rust also provides built-in safety measures and is renowned for its performance as a compiled language. In addition, Rust allows for straightforward and standardized extensibility, which gives the tool room for improvement and future work.

2.1.1 Robustness

Rust has a powerful and intuitive type system that allows for clear and rich data modelling with Structs and Enums. Part of Rust's type system are the Result and Option types; these types allow for error handling and null safety, respectively. The Result type is a wrapper around a value that can be Ok or Err. The Option type is another wrapper around a value that can be Some or None. This allows for smart error handling maximizing Rust's semantic idioms. See below for example.

```
1. fn fallible() -> Result<String, String> {
2.     Ok(String::from("yay"))
3.     // or
4.     Err(String::from("oh no"))
5. }
6.
7.
8. fn nullable() -> Option<String> {
9.     match fallible() {
10.        Ok(val) => Some(val),
11.        Err(e) => {
12.            eprintln!("{e}");
13.            None
14.        }
15.    }
16. }
17.
18. fn main() {
19.     if let Some(msg) = nullable() {
20.         println!("{msg}");
21.     }
22. }
23.
```

Additionally, Rust's compiler is widely regarded as extremely helpful for debugging, as it provides descriptive error messages, helpful examples of error types and thorough documentation. In a case study on Rust as a secure programming language Fulton *et al.* state that 97% of survey respondents 'listed the compiler's descriptive error messages as a major problem-solving benefit' (2021, p. 9). The most important aspect of Rust's robust design is its ownership system, which computes memory operations at compile-time. This prevents most memory-related bugs and vulnerabilities without incurring a performance cost. This was investigated by Bugden and Alahmar, in a paper which concluded that: 'Rust was found to be the safest language compared to C, C++, Java, Go, and Python' (2022, p. 8).

2.1.2 Performance

As well as being a robust language, Rust has also proven to be an extremely performant language. It boasts the use of zero cost abstractions. This concept describes abstractions, such as generic types, that do not induce a runtime cost; however, the term is not entirely accurate, as the cost is seen in longer compile times. The ownership system described above also makes the use of a garbage collector obsolete, which reduces runtime processing even more. Please see Figure 2 for Rust's CPU time benchmarks in comparison to other languages, and Figure 3 for Rust's memory benchmarks in comparison to the same languages.

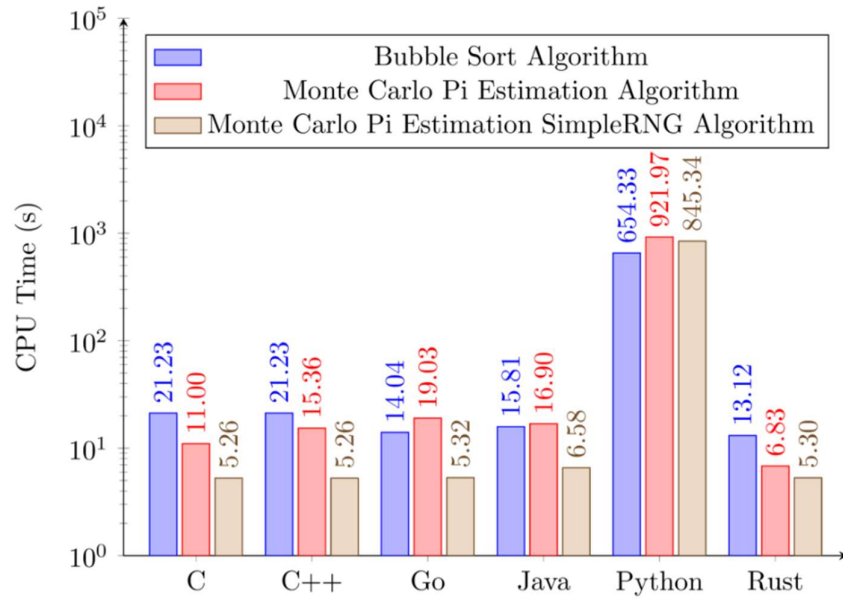


Figure 2: Average CPU time benchmark results (Bugden and Alahmar, 2022, p. 4).

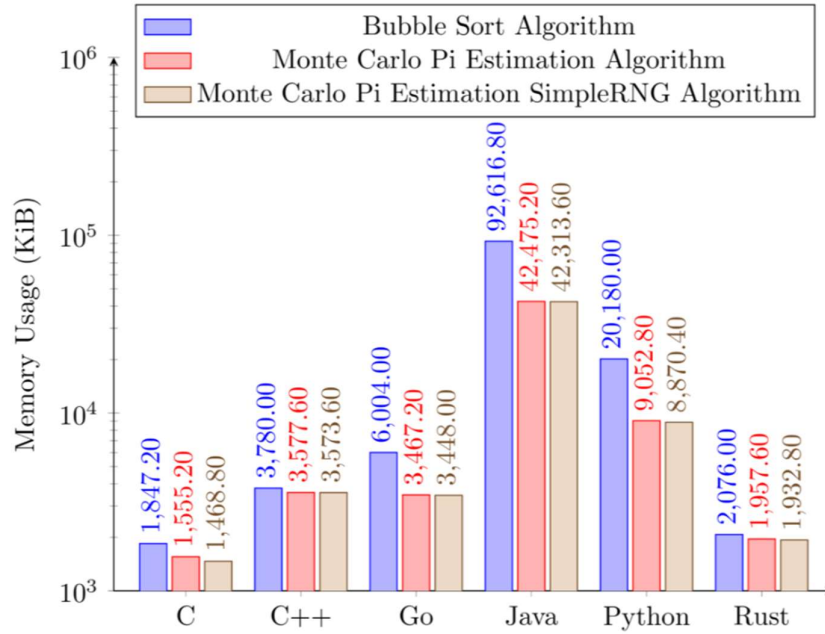


Figure 3: Average memory usage benchmark results (Bugden and Alahmar, 2022, p. 5).

2.1.3 Interoperability

Rust also provides an extensive standard library, with many well-implemented collections, utility functions, types, and traits. Traits are a feature of Rust that allow for shared behaviour across types, similar to interfaces in C++. These allow the programmer to implement traits from different libraries (including the standard library) to their own types and to disambiguate generic arguments. These are used in PEScan to implement serialization and deserialization of data (discussed below in Section 2.2.2), provide sensible defaults for Structs, and accept STDIN and STDOUT for data input and output, respectively (discussed further in Section 2.3.3). The Rust toolchain also includes rustdoc, which is a tool that generates a static documentation webpage from doc comments. This allows all Rust libraries to have a standardized documentation format, which is created with minimal effort. Moreover, the documentation generated for the PEScan project can be used by other developers to quickly understand the codebase and extend the functionality of the tool. Currently, the documentation page is hosted from the GitHub repository of the tool using GitHub pages. For full documentation, see PEScan (2025).

2.2 Dependencies

Rust libraries are colloquially referred to as ‘crates’, PEScan utilizes various crates for five different purposes: robustness, performance, interoperability, network utilities, and data visualization. Separately, the ‘goblin’ crate is used for the main functionality for the program, as it provides binary file header parsing. It is used specifically for parsing the imports header of the given PE sample file. There are a couple alternatives for PE file parsing, although many were not actively maintained (such as ‘pelite’ and ‘libpefile’). The only other viable alternative was the ‘object’ crate, which has considerable abstraction to achieve a single unified I/O interface for many binary file formats; however, it was deemed unfit for purpose as the high level of abstraction causes performance costs and the source of data for the tool only includes information on Windows-specific APIs (mrd0x, no date). The full Cargo.toml metadata can be found in Appendix A.

2.2.1 Robustness

The ‘dirs’ crate is used to locate the platform-specific standard location for the cache directory. This directory is used to store the cached data pulled from *MalAPI.io* (mrd0x, no date). Using standard locations allows the program to be used from any directory in a machine without having to update the cache and can prevent accidental deletion or corruption of the cached data. Furthermore, it prevents accidental overwriting of file with the same name and reduces chances of permission errors greatly.

The ‘anyhow’ crate was selected for error handling instead of ‘thiserror’. The ‘anyhow’ crate provides a standardized Result type with a dynamic error component that can encapsulate any error type. This allows for simpler error propagation from multiple third-party crates that all have their own error types. Generally, this approach is preferred with application code (such as a CLI tool). On the other hand, ‘thiserror’ crate provides a derive macro that implements the standard library’s Error trait, an approach that is instead preferred with library code.

The Rust standard library provides a PathBuf type to handle file paths, however, they are not guaranteed to be encoded in the UTF-8 format. The ‘camino’ crate adds the Utf8PathBuf type, which ensures consistent encoding. Only supporting UTF-8 paths ensures expected behaviour across platforms and allows file paths to be printed for greater error handling and logging capabilities.

For argument parsing, the ‘clap’ crate is the de facto standard for CLI argument parsing in Rust. This is not for nothing; it uses a declarative approach for defining arguments, meaning the CLI interface is strongly typed and well structured. It even includes automatic help (with doc comments) and version (with Cargo.toml metadata) flags. These features are implemented as derive macros, meaning that the code is still clean and readable, whilst preventing runtime penalties. Although, it does cause significant compilation slowdown.

2.2.2 Performance

For reasons discussed in Section 2.3.2, a cache was implemented into the PEScan tool, this meant a format for storing data needed to be chosen. During research, four viable choices emerged: Protobuf, MessagePack, FlatBuffers, and Bincode. Bincode was ruled out due to poor cross-language compatibility, as it was only supported in Rust. The final decision was informed by a study conducted at the University of Houston comparing the performance of serialization libraries (Casey, 2022). Casey concludes that the MessagePack format is both more memory efficient and faster to serialize and deserialize (2022, p. 70). See Figure 4 for output size comparison across a range of input size, and Figure 5 for the combined serialization and deserialization execution time comparison across a range of input sizes.

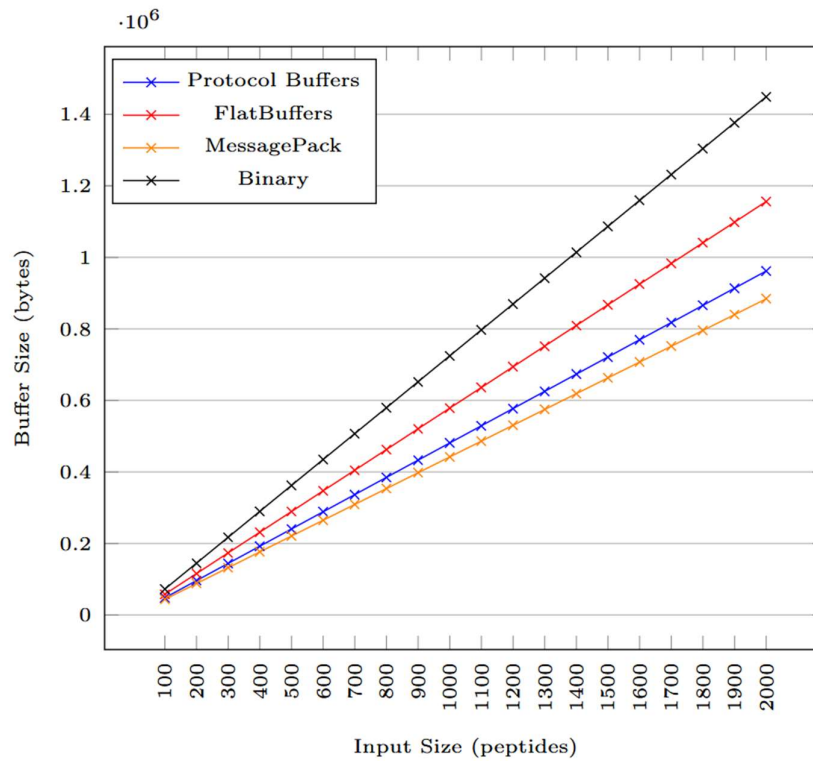


Figure 4: Format comparison of output size across a range of input sizes (Casey, 2022, p. 48)

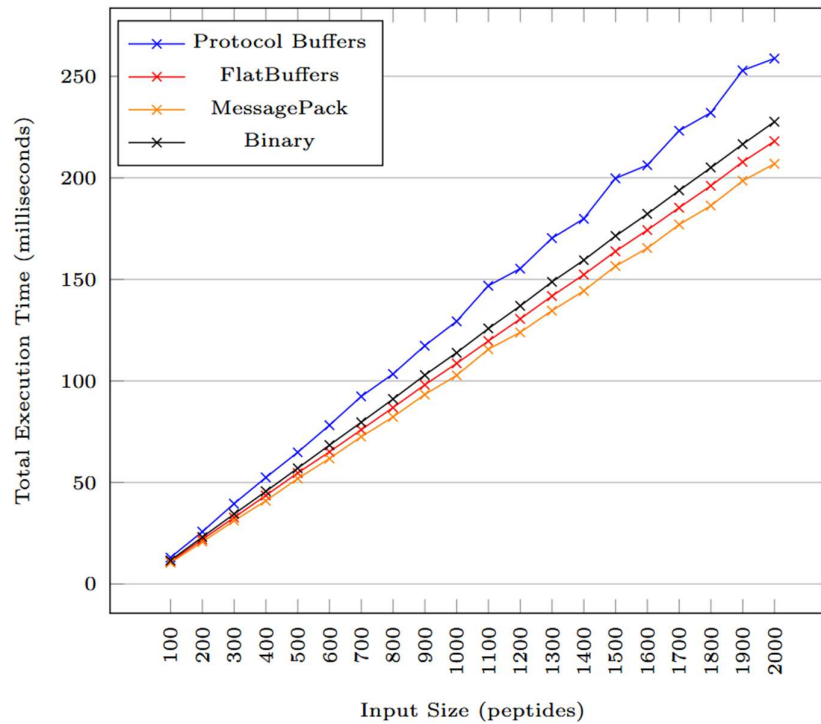


Figure 5: Format comparison of combined serialization and deserialization execution time across a range of input sizes (Casey, 2022, p. 56)

2.2.3 Interoperability

For maximal interoperability, the ‘serde’ crate was used for data serialization and deserialization. The ‘serde’ crate provides the Serialize and Deserialize traits. These traits can be either derived or implemented manually. When a struct implements these traits, they can be serialized and deserialized into any format, provided there is a corresponding Serializer and Deserializer implementation. These implementations are provided by the extensive collection of serde format crates. There is no alternative to the near-infinite extensibility that ‘serde’ provides to implement more output formats. It only takes a couple lines of code, another crate import, and an update to the Format Enum to add another output format. A selection of output formats promotes

interoperability with other software, as clear interfaces enable integration with other tools. The formats implemented as of writing are plain output, JSON, YAML, TOML, and CSV.

2.2.4 Network

Since PEScan needed to perform network requests, asynchronous functions were required. The 'tokio' crate was used as it allows the main function to be async, meaning that async/await syntax in the main function was possible. This made asynchronous code more idiomatic. At some point in development, it was also used for concurrency management, that feature was scrapped (discussed further in Section 2.3.2).

For the actual HTTP requests, the 'reqwest' crate was utilized. The crate provides a reusable Client object which can perform various types of requests. Nonetheless, the tool only performs GET requests, but reusing the client object greatly improves performance. The alternatives were 'ureq', 'surf', 'isahc', and 'hyper'. 'ureq' was ruled out as it is a purely synchronous client, 'isahc' was not native Rust (as it depended on libcurl), and 'hyper' was too low-level. 'surf' was a viable alternative; however, it is a less mature library, and less performant overall (as 'reqwest' is an abstraction over 'hyper').

Since *MalAPI.io* (mrd0x, no date) does not provide a public API (discussed below in Section 2.3), HTML parsing was required. The 'scraper' crate was selected for parsing and extracting data from page HTML. The options were: 'html5ever', 'kuchiki', 'scraper', and 'visdom'. Similarly to 'hyper', 'html5ever' was too low level. 'kuchiki' was unmaintained and seemingly abandoned. 'visdom' was a suitable candidate, but the final choice was influenced by the author's previous experience with 'scraper' as a more robust, mature parser.

2.2.5 Data Visualization

Data visualization was important, as appropriate presentation of data is an essential culminating factor in the usability and practicality of PEScan as a malware analysis tool. During the caching process, a series of progress bars are displayed to provide immediate visual feedback to demonstrate the tool is working as expected; especially since caching runs mostly on the first execution of the tool. For this purpose, the 'indicatif' crate was used for automatically building and displaying the progress bars. This tool is the standard for progress bars in Rust. Additionally, the best format for plain text and terminal output decided was tables. The 'tabled' crate was used to pretty-print tables and even shorten URLs with the OSC-8 ANSI standard (supported by a large amount of terminals). It was picked for its flexibility and aesthetic style.

2.3 Implementation

The tool was inspired by the *MalAPI.io* website (mrd0x, no date), which provides a categorization of Windows APIs into common malware techniques. There are no other comparable data sources, specifically for mapping into malware-specific techniques, such as: spying, ransomware, and anti-debugging. These APIs are categorized by crowd-sourced information through a contribution form that is (presumably) vetted by the maintainer. Unfortunately, there is no public-facing API, codebase, or documentation available, so information gathering was informed using the *robots.txt* file (which imposed no restrictions) and common sense.

2.3.1 Robustness

The reliability of the codebase directly influences the robustness of a tool; thus, native dependencies (such as openssl) and the Rust toolchain were managed using a Nix flake. Nix flakes were chosen as the author developed the program on a NixOS machine and had previous experience using Nix and Nix flakes for software development. Furthermore, the final builds were built as outputs from the Nix flake (see Section 2.3.3 for further detail). Nix allows dependencies to be asserted declaratively and pinned to a specific build of the colossal 'nixpkgs' repository. This approach guarantees complete bit-level reproducibility, which ensures that once the tool builds, it will be able to be built again forever.

The 'anyhow' crate also allowed the author to add context to Results as they were propagating, meaning more descriptive and useful error messages could be applied to fallible functions. In this report, robustness was defined not only as the ability to prevent errors, but also to provide clear and informative error messages. This enables effective correction from the user, which can reduce frustration and can contribute to the maintainability of the application going forward.

2.3.2 Performance

As mentioned above, a cache was necessary in the implementation of this tool. Initially, asynchronous requests to *MalAPI.io* (mrd0x, no date) were made in parallel, all at once. This provided maximum performance; however, it was decided that this was too intense, unnecessary, and discourteous to the maintainer of the website. Hence, a caching system was developed. The requests would be performed sequentially upon the initial execution of the program and stored in the standard cache directory. This improved performance in further executions as there was no need for any further network interactions, so the program was limited only by the efficiency of the implementation itself.

For the data type of the collections used in the deserialized cache and the read PE imports, HashSets were selected. HashSets are functionally the same as HashMaps without a value. They function by hashing the key (with the SipHash 1-3 algorithm for HashDoS protection) and using the hashes within the set to perform lookups, intersections, and other functions in an extremely performant manner. Within PEScan, HashSets are notably used to look up API details from the cache, and to find the intersection of the sample's imported APIs and the cached API names.

2.3.3 Interoperability

For maximum interoperability with other systems and tools, both the input and output of the program can be taken from the STDIN and STDOUT buffers; moreover, aesthetic and error output are printed to the STDERR buffer, which prevents the data from mixing. This encourages interoperability as data can be piped in from another program and can be piped out to another program with ease.

Interoperability is also considered in the implementation of the build within the Nix flake. Outputs are available for compilation to x86_64 Linux and x86_64 Windows. These native executables allow for maximum performance (partly due to the minimal toolchain only for building) on both platforms. It is possible to add more outputs to cross-compile to further platforms (such as ARM); despite that, the flake also outputs a Docker image to run the tool anywhere (albeit at the cost of performance and size).

2.4 Testing

For testing purposes, the 'criterion' crate was added as a development dependency. This precludes any performance impact on the actual tool. This crate is the standard for benchmarking crates in Rust. Alternatively, simpler crates such as 'brunch' or 'bencher' could have been utilized; however, the 'criterion' crate is the only one that provides a 'black_box' function. This function prevents the compiler from optimizing away the target benchmarking function and allows for more accurate benchmarks. A Rust native benchmarking system does exist, but it is only available on the nightly toolchain. This was not suitable for the tool as it was developed with stability as a key tenet.

For memory profiling, a suitable crate was not found. This is because they all required editing of the main source code. The author was not comfortable with having testing functions within the main codebase, as separation of concerns is important for preserving maintainability and reducing performance overhead. Thus, a third-party profiling tool was chosen. *Valgrind's Massif* tool (Nethercote et al., 2025) was chosen for its significantly superior level of detail compared to other options, such as: *Google Performance Tools*, *AddressSanitizer*, and *HeapTrack*.

3 Development

The codebase is separated into four modules: ‘args’ for argument parsing, ‘cache’ for data management, ‘output’ for output formatting, and the main module for the main functionality of the program. The modules were developed sequentially (where possible) and organized by feature. As discussed above, the data management feature originally performed excessive concurrent requests rather than caching data; thus, it was previously named ‘fetch’.

3.1 Argument Parsing

Arguments are defined in the Args Struct and parsed in the main function using the ‘clap’ crate’s derived Parser trait. The critical arguments for PEScan are the sample file positional argument and the update flag. As mentioned before, the sample file argument is optional as the sample can be provided through STDIN. This is useful for passing a sample to a Docker container without having to mount a volume. If neither the sample file argument is provided or the STDIN is a terminal (i.e. nothing has been piped in) the program bails and displays a clear error message. If the update flag is passed, and there is an existing cache file, it is deleted. This forces a re-fetching of the cache from the data source. All arguments can be referred to as their name or their first initial in lowercase (with one exception). See below for example commands.

```
pescan[.exe] ./sample.dll -u
cat ./sample.exe > pescan --update
```

3.1.1 Detail Flags

The detail flags determine which combination of the three possible API details are outputted. The available details are: a summary of the functionality of the API (‘info’), which DLL the API was imported from (‘library’), and a link to the official Microsoft documentation of the API (‘documentation’). Any combination of these flags can be used, and they are all optional. If the program is run with no flags, only the API names will be output. There also exists an alias for all three (the shorthand is an uppercase A). See below for example commands.

```
pescan[.exe] ./sample.exe --info -l
pescan[.exe] ./sample.exe -di
pescan[.exe] ./sample.exe --all
pescan[.exe] ./sample.exe -A
```

3.1.2 Output Flags

The available output flags are used to describe the behaviour of the output data. The width flag only applies to plain text output, as it dictates the width of the tables created with the ‘tabled’ crate. The format flag tells the tool which of the defined output formats to use (discussed in further detail in Section 3.3). The default format used if the flag isn’t specified is plain text. The path flag is an optional Utf8PathBuf (as described in Section 2.2.1). If the path is provided, the data will not be printed to the terminal and will instead be written at the specified file path (creating a new file and failing if a file already exists). One notable exception is the CSV format, which requires the file path to be a directory, for reasons explained in Section 3.3.1.

3.2 Data Management

3.2.1 Cache Structure

The cache is structured using two Structs: a Cache Struct, and an Api Struct. This allows for more granular control over data lookup and storage.

The Cache Struct has two fields: headers, and APIs. These are self-explanatory, but it is crucial to note that the APIs are stored as a Vector of HashSets of Api objects. This is because the cache system leverages the performance capabilities of HashSets to quickly lookup APIs in memory. The reason why a Vector is used as the outer collection is because the order of categories of APIs must be maintained to interface with the parallel array paradigm utilized across the program. The Cache Struct derives three traits: Default, Serialize, and Deserialize. The Default trait is used to construct an empty Cache object and call the ‘update’ method to populate itself. The Serialize and Deserialize traits are needed to store the cache data on the filesystem using the ‘rmp_serde’ crate (Rust MessagePack Serde implementation).

The Api Struct stores the name and details of each API. These are all strings. This Struct derives three traits: Default, Serialize, and Deserialize. The Serialize and Deserialize traits are needed to store the data on the

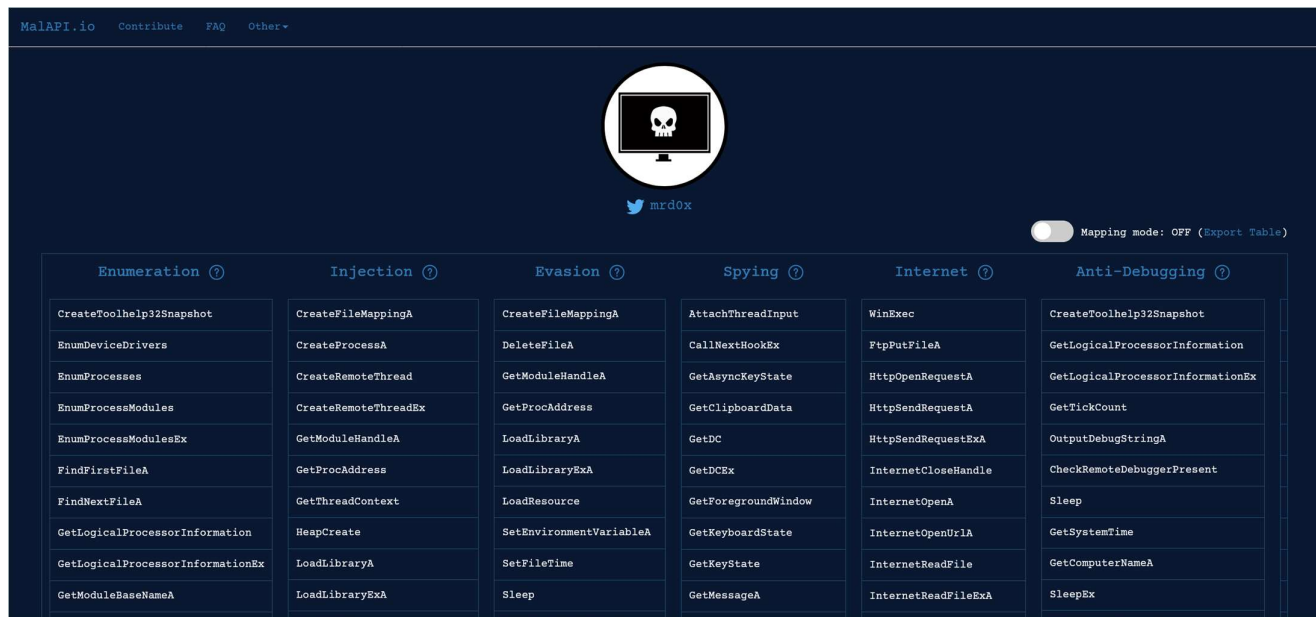
filesystem. The Default trait is used when looking up APIs, explained in further detail in Section 3.2.4. Additionally, three traits are manually implemented: PartialEq, Eq, and Hash. These traits are required to be able to store Api objects in a HashSet as disclosed in the Rust standard library documentation (Rust Project Developers, 2025). These must be implemented manually, as the info, library, and documentation strings are not known when looking up APIs. Thus, only the name is used for the PartialEq and Hash implementation. These traits are implemented with the following property, as defined in the HashSet documentation (Rust Project Developers, 2025). Given Apis a , b and hash function H :

$$a = b \rightarrow H(a) = H(b)$$

The Eq trait does not include any methods, because it is only a property indicator that equivalency operations are reflexive.

3.2.2 Index Scraping

The basic data required for the tool's basic functionality are found in *MalAPI.io*'s index page (mrd0x, no date). This data is presented in the form of a table with APIs under a set of headers. This information can be gathered with a single HTTP GET request, while being parsed and collected with the 'scraper' crate. See Figure 6 for the index page at the time of writing.



Enumeration	Injection	Evasion	Spying	Internet	Anti-Debugging
CreateToolhelp32Snapshot	CreateFileMappingA	CreateFileMappingA	AttachThreadInput	WinExec	CreateToolhelp32Snapshot
EnumDeviceDrivers	CreateProcessA	DeleteFileA	CallNextHookEx	FtpPutFileA	GetLogicalProcessorInformation
EnumProcesses	CreateRemoteThread	GetModuleHandleA	GetAsyncKeyState	HttpOpenRequestA	GetLogicalProcessorInformationEx
EnumProcessModules	CreateRemoteThreadEx	GetProcAddress	GetClipboardData	HttpSendRequestA	GetTickCount
EnumProcessModulesEx	GetModuleHandleA	LoadLibraryA	GetDC	HttpSendRequestExA	OutputDebugStringA
FindFirstFileA	GetProcAddress	LoadLibraryExA	GetDCEX	InternetCloseHandle	CheckRemoteDebuggerPresent
FindNextFileA	GetThreadContext	LoadResource	GetForegroundWindow	InternetOpenA	Sleep
GetLogicalProcessorInformation	HeapCreate	SetEnvironmentVariableA	GetKeyboardState	InternetOpenUrlA	GetSystemTime
GetLogicalProcessorInformationEx	LoadLibraryA	SetFileTime	GetKeyState	InternetReadFile	GetComputerNameA
GetModuleBaseNameA	LoadLibraryExA	Sleep	GetMessageA	InternetReadFileExA	SleepEx

Figure 6: *MalAPI.io* index page (mrd0x, no date)

When viewed as source, it was revealed that the data is stored as a series of nested tables. The headers are the only table header tags, whilst the APIs were displayed with table data tags (padded with a seemingly random amount of empty table row tags). Additionally, all the APIs had a class of 'map-item'. The tool uses two methods in the Cache Struct to scrape this data, one for the headers, and one for the APIs. These are defined as 'scrape_headers' and 'scrape_apis', respectively.

The header scraping method simply uses the 'scraper' crate's Selector Struct with the CSS selector 'th' to gather all the header tags. It then uses Rust's 'map' combinator in conjunction with the 'scraper' crate's convenient 'text' method (which extracts all text from selected elements recursively into an iterator) to trim the header strings and collect them into a Vector of strings. This data is then stored in the Cache's 'headers' field.

The API scraping method is more complicated, as it must maintain the APIs in their defined categories. The first selector is used to gather all the columns' table body tags. The second selector is used to gather all elements with the 'map-item' class within these columns. See below for the defined selectors.

```
1. let column_selector = Selector::parse("td > table > tbody")
2. .map_err(|e| anyhow!("failed to parse selector: {e}"))?;
3. let api_selector = Selector::parse(".map-item")
4. .map_err(|e| anyhow!("failed to parse selector: {e}"))?;
```

To gather all the APIs, the 'for_each' combinator is used with the column iterator (generated using the 'select' method) to push each column's APIs as a Vector of strings to a 2D Vector. These APIs are extracted in the same way the headers were extracted (by mapping the 'text' method to the API selector's iterator). It is important to note that in contrast to the header scraping method, whose Ok variant returns a unit type (comparable to a null type), the API scraping method returns the actual data, instead of storing it in the Cache Struct. The reasoning for this is made clear in the next section.

3.2.3 Details Scraping

To fetch the details which can be provided (using the detail flags shown in Section 3.1.1), multiple requests need to be made. This is because these details are exposed within individual pages for each API. These pages are all under the '/winapi' URL namespace. The logic for fetching these details is defined as the 'update' method of the Cache Struct. See Figure 7 for an example details page.

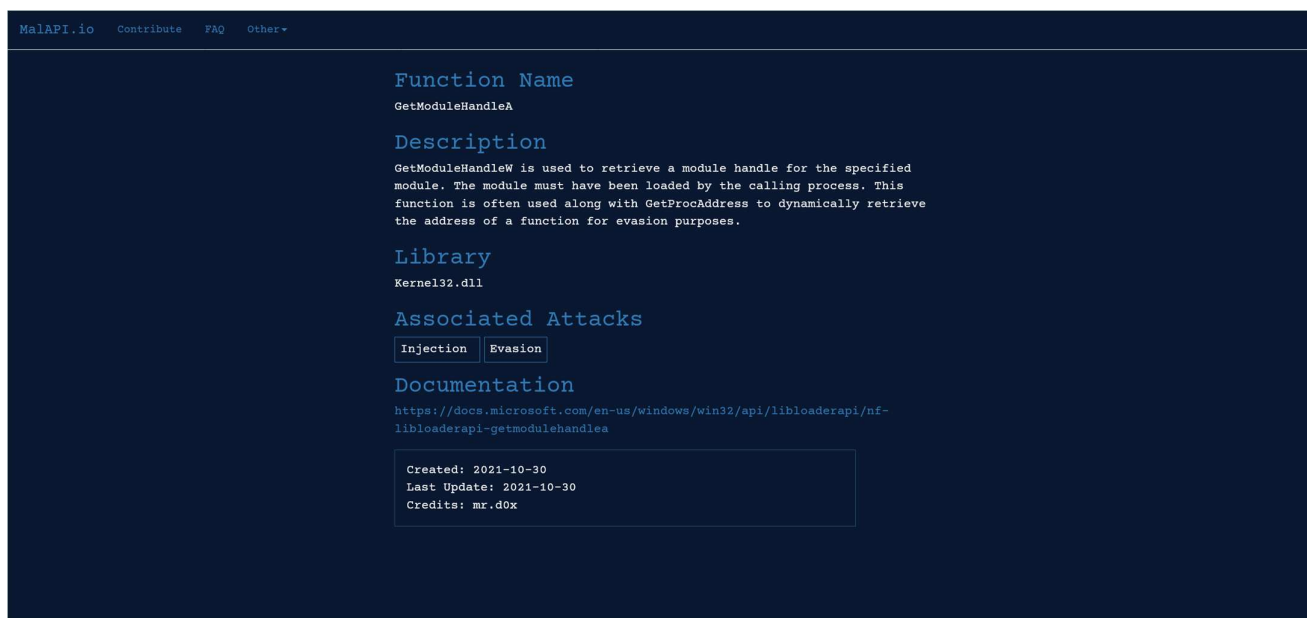


Figure 7: MalAPI.io details page for GetModuleHandleA (mrd0x, no date)

When viewing the source for this page, it could be seen that each piece of information is stored within a div tag with the class 'detail-container'. Furthermore, the actual data is displayed within a div tag with the class 'content'.

First, the 'reqwest' client is set up with a user agent header (derived from the tool name and version), the base URL for the details pages is defined, and the CSS selector 'content' is parsed. Next, the index page is fetched and parsed for header and API collection. The previously discussed header and API scraping methods are called. The APIs are not stored within the Cache Struct because the Cache only stores a collection of Api Structs (denoted in Section 3.2.1). The APIs Vector is used within the update method to fetch the details, and the API name is subsequently included in the generated Api objects. With that in mind, the APIs Vector returned by the API scraper method will be referred to as the API name Vector from now on. The Cache's APIs field is initialized with the size of the API name Vector and the API names are then looped through by fetching an iterator of references and enumerating it to be able to display headers for the progress bars. For each category of APIs, a progress bar is displayed with the category name and which API is currently being fetched. An empty HashSet is then pushed to the Cache's APIs field, while also reserving space in memory with the length of the category. Each API in the category is looped through and the details page is fetched from MalAPI.io (mrd0x, no date). The status of the request is then verified to be OK; if it is not OK, then a message is displayed with the header and API name above the progress bars, an incomplete Api object is inserted into the category HashSet, and the update continues to the next API. If it is OK, then the page is parsed, and the details are selected and collected into a Vector of strings. Using the page source as reference, it was determined that the 'info' detail could be found in the second element (first index), the 'library' detail could be found in the third element (second index), and the 'documentation' detail could be found in the fifth element (fourth index). An Api object is then constructed and inserted into the category HashSet. After each successful scraping, the progress bar is incremented.

3.2.4 Deserialized Cache Access

The first of the deserialized cache access methods is the ‘get_api’ method, which is used to lookup an API by name and category index and fetch its details. It is a relatively simple method; it first constructs a ‘lookup’ Api object using the partial Struct initialization Rust pattern. See below for the lookup object implementation.

```
1. let lookup = Api {  
2.     name: name.to_owned(),  
3.     ..Default::default()  
4. };
```

It then uses the category HashSet’s get method to retrieve an Option containing the full object if it exists, or None otherwise.

The second method is the ‘get_apis’ method. This method is used to generate an API name Vector. This is necessary for cross-referencing with the sample API imports (detailed in Section 3.4.2). The method uses two nested map combinators instead of the ‘flat_map’ combinator because it needs to preserve the separate category HashSets. The names are cloned to give ownership to the calling function, this is needed to calculate the intersection HashSet with the imported APIs.

3.2.5 Serialized Cache Access

The logic for serialized cache access is contained within the ‘load’ method of the Cache Struct. This function deals with loading the cache from disk, creating it if it doesn’t exist, or re-creating it if the update flag is set. First, a default Cache object is created. As explained in Section 3.2.1, this is to be able to call the update method for self-population. First, the method checks if a cache directory is accessible. If not, the cache is updated but not stored on disk; if it does, the cache file path is constructed. Then, it checks for the update flag. If the cache exists and the update flag is set, the file is deleted from disk, forcing the update method to run. If the cache file exists but the update flag is not set, the file is read from disk and deserialized using the ‘rmp_serde’ crate. If the cache file doesn’t exist (or has been deleted because the update flag is set), the PEScan-specific cache directory path is retrieved, the update method is called, the necessary directories are created, and the cache is written to disk. Finally, unless an error occurs and the function is returned early with an Err object, the Cache object is returned.

3.3 Output Formatting

The plain text output of the program is created using the ‘tabled’ crate. This is done in two parts. The first part is adding derive macros for the SuspectImport Struct (which is described in the next section). The Struct must derive the Tabled trait, and optional attribute macros are used to set a default behaviour for details that are None (an empty string). Furthermore, the documentation detail has an attribute macro that points to a custom ‘format_url’ function. This function checks if the field has a value, and if it does it shortens the URL using the OSC-8 standard ANSI escape sequence (egmontkob, 2025); otherwise, it returns an empty string (in line with the other details’ behaviour).

The tables are created using the ‘create_tables’ function. This function returns a Vector of pairs of strings and Table objects (provided by the ‘tabled’ crate). First, the return Vector is initialized with a suitable size, then each Table object is created (one for each category). Using the library interfaces provided by the ‘tabled’ crate, the Table object is generated from each category Vector. Then the number of columns necessary for each table is computed and columns are removed if necessary. Finally, wrapping is applied to every row, in accordance with the width flag and the tables and headers are returned.

3.3.1 Data Models

The first model is the Format Enum, it is marked as non-exhaustive with an attribute macro. This means that the Enum is likely to be expanded in the future (with more formats) and external programs who wish to use the PEScan modules are forced have a wildcard match arm when matching to this Enum, to allow for format additions to not be breaking changes. At the time of writing, this Enum contains the following formats: TXT, JSON, YAML, TOML, and CSV. This Enum also derives the ValueEnum trait, which allows it to be used as a robust, restricted value set for the format flag.

The Details Struct is used to enable optionality for detail output. This Struct allows for the program to discern which details should be included in the output or not. This is only used for internal logic, as the tables would not display properly if the SuspectImport Struct contained a Details object field instead of individual detail fields.

The SuspectImport and Output Structs are the key data models that allow for the serialization of data into multiple formats. The Output Struct contains a Vector of headers and a 2D Vector of SuspectImport objects, which is essentially the same as the Cache Struct. The only difference is that the inner collection of the Cache Struct is a HashSet (for the reasons discussed in Section 3.2.4), and the Output Struct uses a Vector. This is because the Output Struct has no need for fast lookups and comparisons, and a Vector is more memory efficient. Even so, the important part of the Output Struct lies in its implementations. It has a custom implementation for the Serialize trait, which is necessary to pair the headers with the API data. This is done with the ‘serde’ crate’s ‘serialize_map’ serializer method. The implementation performs a zip operation between iterators of its ‘headers’ and ‘suspect_imports’ fields and loops through the resulting iterator to use each header as a key for each category. In addition, empty categories are not serialized at all. Finally, the inherent implementation for the Output Struct provides the methods for serializing into the supported formats. All format methods are implemented in largely the same manner (with one exception): they take a generic argument with a Write trait bound. This allows the methods to treat the argument as a writable buffer and enables writing to STDOUT and to files with the same method implementation. The method then serializes itself with the various format crates and writes to the buffer.

The exception is the CSV format. This is because the CSV format has no way to serialize nested data; thus, the data must be output to either multiple files or headings must be displayed before the serialized data in STDOUT. This is why the path flag for the CSV format must be a directory. Similarly, the serialization methods need to be split into STDOUT and file output. This is because the path argument must be validated as a directory if outputting to files, or the headers need to be printed out if outputting to STDOUT. Despite that, they are implemented in generally the same way. The table headers are constructed depending on the flags set, and the individual SuspectImport objects are serialized into each row. If a SuspectImport’s details are incomplete (from an unreachable details page), the serialization would fail (as the CSV format requires all rows to have the same number of columns). The failure would trigger the method to pad the row with extra empty fields.

The SuspectImport Struct contains functionally the same fields as the previously discussed Api Struct, but with Option types and an attribute macro that tells the ‘serde’ crate not to serialize fields that are None. The ‘len’ method is implemented in the Struct to allow the CSV serialization methods to calculate the number of empty fields needed to pad an incomplete row. It is important to know that despite the SuspectImport and Details Structs having identical fields, there is no data duplication between them (and the API name Vector). This is achieved by having the SuspectImport Struct store a reference to the corresponding API name and Details object. This is made possible with Rust’s lifetime parameters, allowing the references to live for the scope of the Output object. The headers Vector is not a reference; however, data is still not duplicated as ownership of the variable is simply transferred from the Cache Struct to the Output Struct. See below for the Output and SuspectImport Structs’ definitions (omitting derive and attribute macros).

```
1. pub struct Output<'b> {
2.     /// [Vec] of technique categories
3.     pub headers: Vec<String>,
4.     /// 2D [Vec] of suspect APIs by technique category
5.     pub suspect_imports: Vec<Vec<SuspectImport<'b>>>
6. }
7.
8. pub struct SuspectImport<'a> {
9.     /// Name of API
10.    pub name: &'a String,
11.    /// Summary of API functionality
12.    pub info: Option<&'a String>,
13.    /// Library from which API is imported
14.    pub library: Option<&'a String>,
15.    /// Link to API documentation
16.    pub documentation: Option<&'a String>,
17. }
```


3.4 Main Functionality

3.4.1 Input

First, the initial data is declared. The arguments are parsed into an Args object (as defined in Section 3.1), the cache is loaded, and the API name Vector is extracted (as described in section 3.2). The sample is read into a Vector of bytes either from the provided sample argument or from STDIN. If neither is available, the program bails with an error message. The buffer is then parsed with the ‘goblin’ crate’s Object Struct. If there is an error parsing (but not reading the buffer) it is most likely that the program is being run from Docker without interactive mode enabled. The build script for the Docker image sets an environment variable to allow for a descriptive and accurate error message for this case. Otherwise, it must be some other parsing error. This Object is then matched with the various formats supported by the ‘goblin’ crate. Any files that are not binary objects still parse successfully, but they are caught by the wildcard arm and the program bails with an invalid filetype error. The imports are extracted from the PE Object and flattened using the ‘flatten_import’ function. This function removes all the extraneous data provided by the ‘goblin’ crate and returns a HashSet of strings. Two collections are then initialized to hold the matched imports and their corresponding details. These are named ‘suspicious_imports’ and ‘details’ respectively. The PE Object is then dropped from memory as it is no longer used. Note that this function takes ownership of the object before dropping it, preventing use-after-free errors.

3.4.2 Processing

Next, an intersection iterator is calculated between each category HashSet within the API name Vector and the extracted imports. The iterator’s elements are cloned (to own the values) and collected into the suspicious imports Vector. Then, a 2D Vector of Details is then constructed and with values extracted from the cache, depending on the detail flags set. Again, the values are cloned. If no details are found, the default value for Details is added (all None).

3.4.3 Output

All these values are then gathered (as references) in a 2D Vector of SuspectImport objects. With this Vector and the cached headers, the Output object is created. The format flag parsed in the arguments is then matched. The first match is a guard arm to prevent the unnecessary creation of a heap-allocated writer when the format is a CSV, as the CSV serializing methods do not take a writer (as explained in Section 3.3.1). For all other formats, a heap-allocated writer is created using the following logic.

```
1. let mut buf: Box<dyn Write> = if let Some(path) = &args.path {  
2.   Box::new(fs::File::create_new(path)?)  
3. } else {  
4.   Box::new(std::io::stdout())  
5. };
```

A Box in Rust is a pointer that stores a value on the heap. This is necessary because the ‘buf’ variable could be either a File or Stdout type. A nested match arm then matches the other types to their corresponding serializing methods. In Rust matching an Enum requires all variants to be matched, thus a CSV arm is added; nevertheless, it uses Rust’s unreachable macro to denote it as unreachable (as it would be matched in the parent match construct). Finally, a credit to mrd0x and contributors is printed to STDERR (as it shouldn’t be included when redirecting output).

3.5 Benchmarking

As mentioned in Section 2.4, ‘criterion’ was used to perform accurate and informative benchmarks. These were implemented in the ‘benches/full.rs’ file. The ‘bench’ setting was disabled for the tool, to indicate to the Rust compiler to leave the benchmarking processes to ‘criterion’. The entire program was benchmarked by using the Rust standard library’s Command Struct to spawn the process. Along with this baseline benchmark, several benchmarks with all the possible format flags and the ‘all’ alias flag were tested. The ‘criterion’ crate runs all these benchmarks in the same way. The process is warmed up to ensure reliable timing results, this is needed to mitigate first-run effects such as ‘cold’ caches. On top of that, it allows the ‘criterion’ crate to estimate execution times and calculate meaningful sample sizes. See Appendix B for full benchmark code.

4 Results

4.1 Robustness

PEScan aimed to be robust through effective error propagation and clear error messages. This was achieved, since the program does not panic, and instead errors bubble up through methods and functions and are handled with the 'anyhow' crate. Useful error messages are actualized through attaching context to Result types. This is shown through various error messages. When a sample is not provided through STDIN or through the positional sample argument, error message is shown.

```
C:\Users\user\Desktop\Samples\5>pescan-x86_64-windows.exe
Error: sample not found in [FILE] or stdin.
```

Figure 8: Sample not found error

When the program is run through Docker, and a file is piped through STDIN, but the interactive mode is not enabled, the following error message is shown.

```
2025-04-29 20:13:58
2025-04-29 20:14:50 Error: docker container not running in interactive mode
2025-04-29 20:14:50
2025-04-29 20:14:50 Caused by:
2025-04-29 20:14:50     Malformed entity: Object is too small.
```

Figure 9: Docker interactive mode error

When the filetype of the sample file is incorrect (despite file extension), the following error message is shown.

```
C:\Users\user\Desktop\Samples\5>pescan-x86_64-windows.exe sample.exe
Error: invalid file type, only PE files are supported

C:\Users\user\Desktop\Samples\5>
```

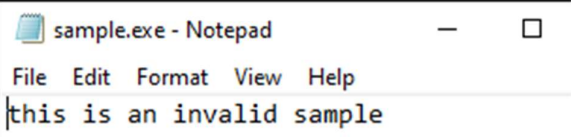


Figure 10: Invalid filetype error

When the cache is corrupted, the following error message is shown, suggesting a cache update to fix the corruption.

```
C:\Users\user\Desktop>pescan-v2.0.5-x86_64-windows.exe
Error: corrupted cache, run with --update to fix

Caused by:
    invalid type:integer `101`, expected a sequence
```

Figure 11: Corrupted cache error

The argument system is also robust by nature, as the 'clap' crate handles argument parsing internally. This applies to both non-existent arguments and incorrect arguments.

```
C:\Users\Diego Andrade\Documents\CMP320>pescan-v2.0.4-x86_64-windows.exe --nonexistent -o
error: unexpected argument '--nonexistent' found

  tip: to pass '--nonexistent' as a value, use '-- --nonexistent'

Usage: pescan-v2.0.4-x86_64-windows.exe [OPTIONS] [FILE]

For more information, try '--help'.
```

Figure 12: Non-existent argument error

```
C:\Users\Diego Andrade\Documents\CMP320>pescan-v2.0.4-x86_64-windows.exe -f
error: a value is required for '--format <FORMAT>' but none was supplied
[possible values: txt, json, yaml, toml, csv]

For more information, try '--help'.
```

Figure 13: Invalid argument error

In the same vein, when the CSV output path is not a directory, the following error is displayed.

```
C:\Users\user\Desktop>pescan-v2.0.5-x86_64-windows.exe sample.exe -f csv -o not_a_dir.csv
Error: csv format requires output path to be directory
```

Figure 14: Invalid output path for CSV error

Apart from this, due to the nature of Rust and the way the tool has been implemented, it is almost guaranteed that memory allocation related vulnerabilities and errors are not present. This is due to the Rust's ownership system, which was described in Section 2.1.1.

4.2 Performance

The benchmarks were performed through 200 iterations, with a sample size of 100. The 'criterion' crate displays results with a confidence interval. The confidence level was left at the default value of 95%. To reiterate, the benchmarks performed were as follows: a baseline (no flags) and every available format with the 'all' flag enabled. To evaluate the performance of the program, the 'criterion' crate provides both mean (with standard deviation) and median (with median absolute deviation) values. Based on the distribution of the benchmark data, the usage of mean and standard deviation to evaluate performance. The data showed a relatively normal distribution, and the author was confident there was a low amount of benchmark noise. See the following graph for the distribution of the baseline benchmark and Appendix C for the distribution graphs of all the benchmarks.

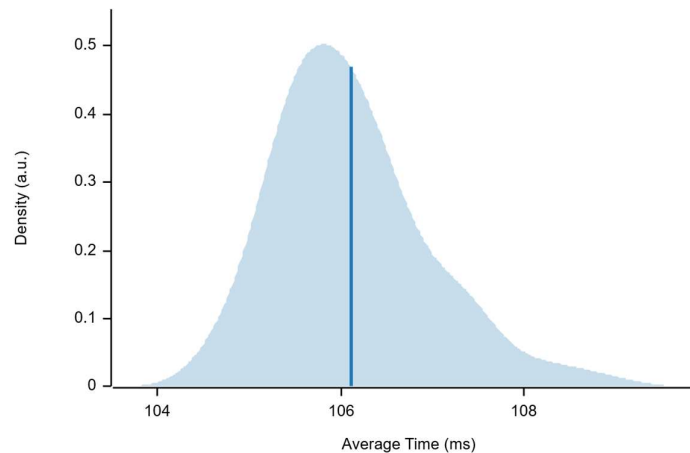


Figure 15: Distribution of baseline benchmarks

The following table presents each benchmark's estimated execution time, amount of outlier samples, and the confidence intervals for the mean, and standard deviation.

Name	Estimate (ms)	Outliers	Mean (ms)	Standard Deviation (µs)
Baseline	106.1	3	[106.0, 106.2]	[655.0, 957.0]
By format ('all' flag set)				
TXT	107.3	1	[107.1, 107.4]	[655.7, 903.5]
JSON	103.2	0	[103.0, 103.3]	[686.6, 866.4]
YAML	103.3	3	[103.2, 103.5]	[694.1, 953.1]
TOML	103.2	0	[103.1, 103.3]	[631.4, 804.6]
CSV	103.2	1	[103.1, 103.4]	[588.9, 797.8]

Table 1: Benchmark measurements to 4 significant figures

All benchmarks were run on the following system:

- **CPU:** Intel i9-11900H running at 2.5GHz
- **RAM:** 32GB DDR4 running at 3200MHz
- **Storage:** SSD NVMe
- **OS:** NixOS version 24.05 (64-bit)
- **Rust:** version 1.85.1 (stable)

The estimate time shows that the difference between running PEScan without detail flags and with detail flags (outputting to plain text) is around 1.2ms. Similarly, the difference between plain text (with the ‘tabled’ crate) is slower than the other formats (with the ‘serde’ crate) by around 4.1ms. The stability of the run-time of the tool can be measured with the standard deviation column. The estimated standard deviation can be calculated with the following formula, given σ_L is the lower bound, and σ_U is the upper bound:

$$\sigma = \frac{\sigma_L + \sigma_U}{2}.$$

With this value, it is possible to calculate the mean standard deviation across all benchmarks. This value was computed as: 766.2 μ s. The mean estimated execution time across all benchmarks was computed to be: 104.4ms. This meant that the mean estimated variation across all benchmarks was: 0.7339%. This indicates that the benchmarks were consistent throughout.

As mentioned in Section 2.4, the *Valgrind Masiff* tool (Nethercote *et al.*, 2025) was used to measure memory usage. Two commands were analysed: sample analysis with the ‘all’ flag with the TXT output format, and sample analysis with the ‘all’ flag in the JSON output format. This was to compare the difference between serializing data with the ‘tabled’ crate compared to the ‘serde’ crate. See the following graphs the detailed memory usage graphs, and Appendix D for more readable full-size graphs.

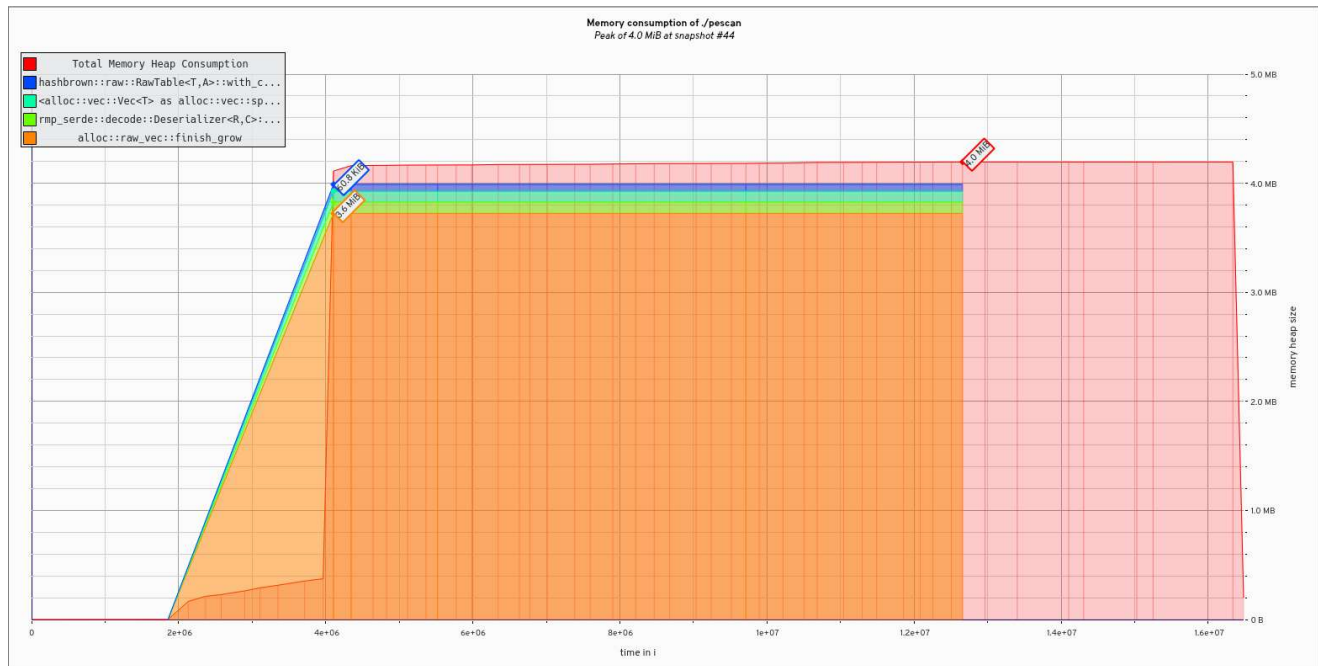


Figure 16: Memory consumption of PEScan with all details in the TXT format

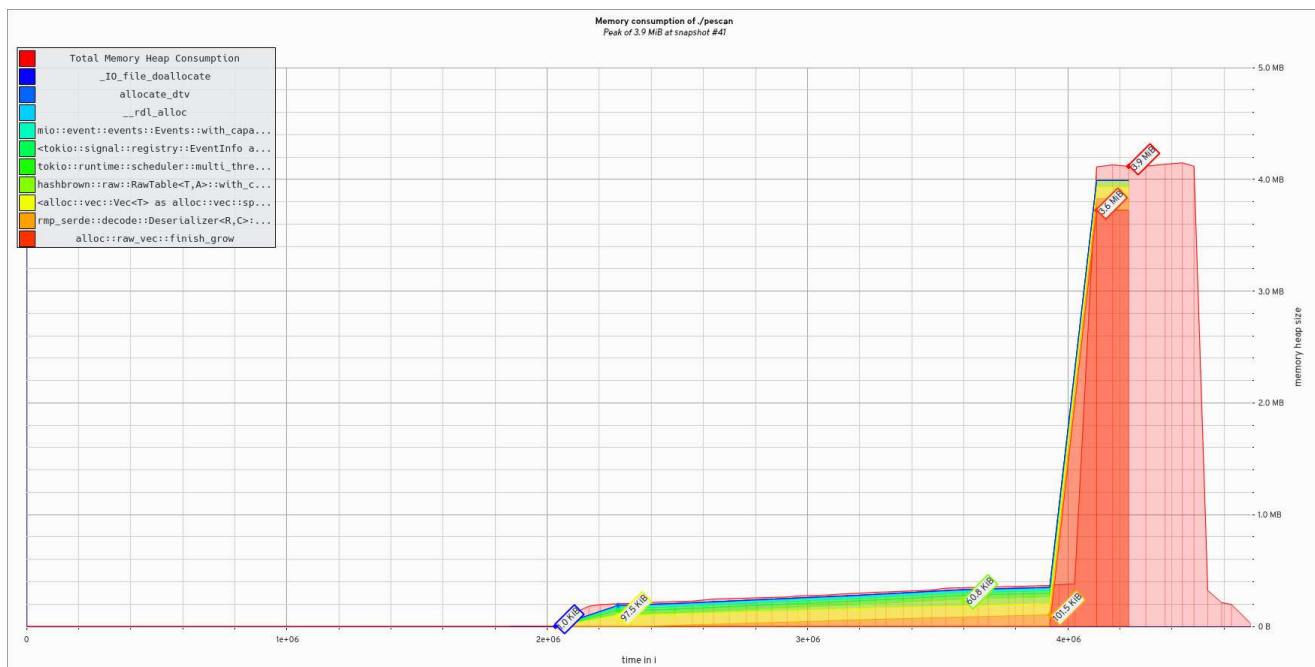


Figure 17: Memory consumption of PEScan with all details in the JSON format

It is plain to see that the ‘serde’ crate is far more optimized and uses memory for less amount of time. Interestingly, there is only a 100KB difference in peak memory usage (4MB compared to 3.9MB). By examining the legend, it is seen that the bulk of the memory usage comes from the ‘raw_vec’ allocator. This makes it clear that the bulk of memory usage comes from the stored Vector and HashSet variables. Ultimately, it is shown that there is a direct correlation between memory usage and execution time.

4.3 Interoperability

PEScan allows for seamless interoperability in three forms: extensibility of the codebase, multiple binary releases, and support for external tool integration. The tool was developed in a modular format, this allows easy extension and maintaining of the code. Additionally, the ‘rustdoc’ Cargo tool allowed for the automatic generation of documentation. This allows other developers to understand the codebase and quickly start implementing extensions if desired. See Figure 18 for screenshot of the documentation index.

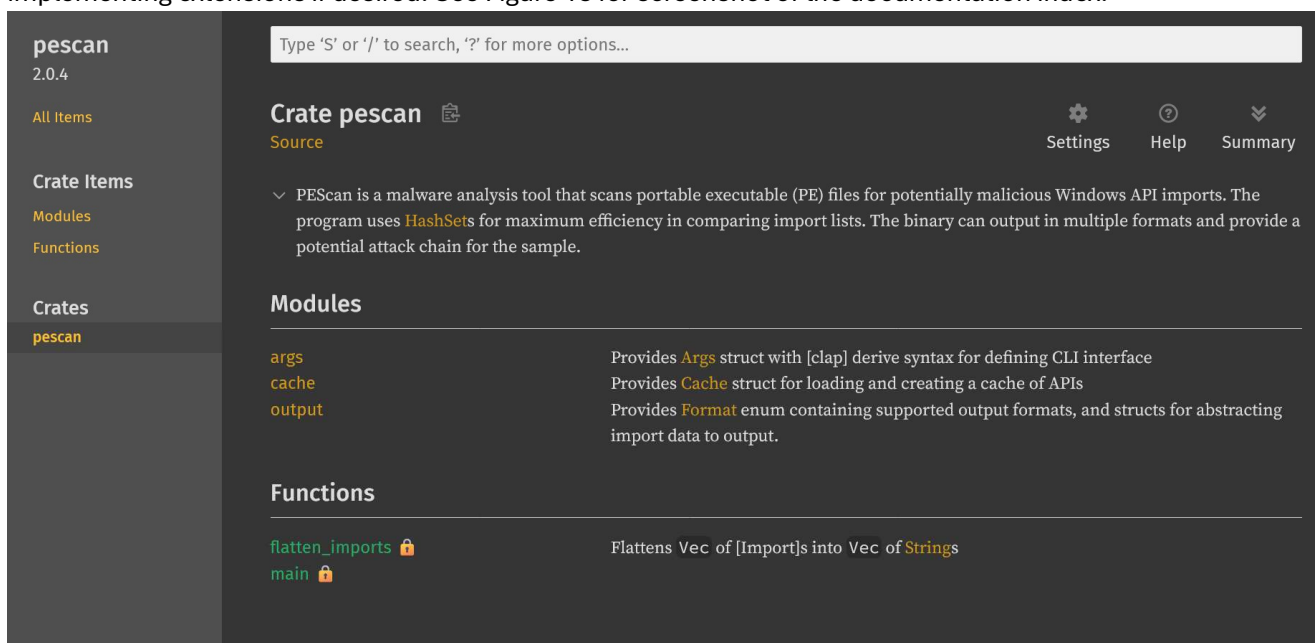


Figure 18: Screenshot of documentation index page

The PEScan tool provides three different built binaries, two native builds and one Docker image; all built through a Nix flake. The built PEScan binaries are sized as follows:

- **Docker:** 20MB
- **Windows 64-bit:** 12.3MB
- **Linux 64-bit:** 7.26MB.

Practically, this means that the Docker build will be less performant; however, it can be run on virtually any environment. Trading performance for compatibility is a common trade-off in software development.

4.3.1 External Tool Integration

As discussed in Section 3.3.1, PEScan supports integration with external tools through appropriate use of STDIN, STDOUT, and STDERR streams. This allows for seamless integration with any number of CLI tools, such as: *awk*, *sed*, *jq*, *grep*, and more. See below for an example of integration with the *jq* command-line JSON processor.

```
> ./pescan-v2.0.5-x86_64-linux sample.exe -lf json | jq '["Ransomware"]'
Data provided by mrd0x & contributors via https://malapi.io.
[
  {
    "name": "CryptGenRandom",
    "library": "Advapi32.dll"
  },
  {
    "name": "CryptAcquireContextA",
    "library": "Advapi32.dll"
  }
]
```

Figure 19: Example of integration with *jq*

4.4 Comparison with Existing Tool

The only comparable tool found was *MalAPIReader* (Squiblydoo, 2024), it was implemented in Python and included a caching feature (called LocalStorage). It provides less granular control over output, and only has one output format. Due to the program being written in Python, the benchmarking was performed using the ZSH ‘time’ command. With a small sample set with all details enabled (‘verbose’ flag) and the same sample file, a mean execution time was found to be 218.3ms. This is more than twice the comparable PEScan benchmark. Finally, memory profiling was conducted using the same tool. See below for the memory usage graph of *MalAPIReader* (Squiblydoo, 2024), and Appendix D for the full-size graph.

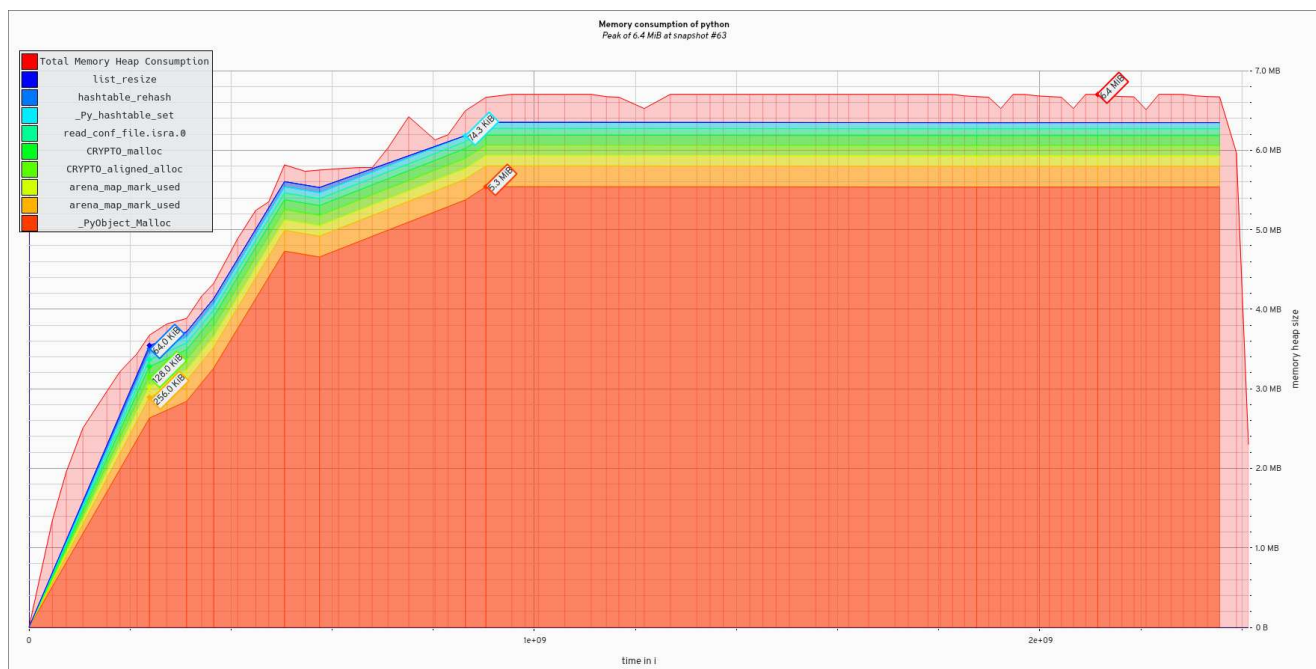


Figure 20: Memory consumption of *MalAPIReader* (Squiblydoo, 2024)

This shows that the peak memory usage was 6.4MB and it is allocated for a longer period.

5 Discussion

PEScan was developed with three key factors in mind: robustness, performance, and interoperability. These aims were met. The tool was developed using a variety of data structures and dependencies best suited to give a balance of performance and extensibility. Moreover, the program was designed for graceful error propagation with precise and effective error messages.

5.1.1 Robustness

The tool stores its data in a dynamically generated cross-platform standardized location, which allows for maximum compatibility with systems and easy troubleshooting. Even more importantly, errors are handled in a manner that provides a rich and exhaustive context for all error types. The ‘anyhow’ crate displays a backtrace of errors, beginning with the custom context attached by the author. This allows end-users to be able to benefit from Rust’s well designed error system. End-users also benefit from knowing that memory related issues and vulnerabilities are not an issue for this program, as they will have been caught by the compiler during development. In their case study on Rust, Bugden and Alahmar suggest: “ensuring memory safety can eliminate up to approximately 70% of security bugs”.

5.1.2 Performance

PEScan achieves its performance goals through two key strategies: leveraging Rust’s design paradigms and standard library and implementing an efficient cache. Rust helped make the tool performant through the ownership and lifetime systems, which allowed the author to safely pass data between scopes while minimizing data duplication. This is seen with the minimal 1.2ms slowdown when outputting details. The Rust standard library provided a high-performance HashSet implementation that allowed the program to cross-reference 8 large lists with the PE imports in barely more than a tenth of a second. The cache implementation utilized the MessagePack format to store a small and quick to parse file on disk. HashSets are utilized here as well to provide performant detail lookups from the deserialized cache. All these factors allow PEScan to outperform the existing tool, whilst also providing more granular output control and greater interfacing capabilities. There are not enough malware analysts for the number of malware which is released year on year; however, a performant static analysis tool can save time and help with workload.

5.1.3 Interoperability

The tool promotes interoperability through multiple output formats, effective output stream usage, and well-defined data models. The ‘serde’ crate allows for a generic serialization implementation, which can be used with an expansive catalogue of format crates. Having multiple output formats (that can output to STDOUT or to a file) presents the end-user with a unified output channel that can interface with any number of external tools or libraries. This was demonstrated in Section 4.3.1, by processing JSON output with *jq*. This extremely powerful behaviour can be used to further automate (and thus speed up) the static analysis process. PEScan is also extensible through the codebase. Transparent data models allow developers to capitalize on the modular nature of the tool and further improve and introduce additional functionality. In a more indirect way, intuitive data visualization (such as tables and shortened links) can improve analyst productivity; and external tools like *awk* and *sed* can be used to customize and tailor data output.

5.1.4 Limitations

There are two areas with room for improvement. The first is the ‘tabled’ crate’s lack of performance. Compared to the ‘serde’ crate’s performance in both the benchmarks and memory profile, the implementation is less optimized, and the interface is unintuitive. The design seemed convoluted, largely because the documentation was lacking. The second area for improvement is the structure of the data models. The structure contains a lot of duplicate fields (which do not affect performance, as explained in Section 3.3.1), which makes seem redundant. It was designed as it is due to the strict CSV serializing requirements and the ‘tabled’ crate’s lack of Struct inlining feature.

6 Future Work

Some improvements could be made to PEScan's output module. This could include a custom plain text table serializer, possibly utilizing the 'serde' crate, as this would both improve performance and resolve redundancy in the data models; however, this was outside the report's scope. This tool could also be expanded upon by analysing more data from a PE file's headers or extending the current functionality to more binary formats. A further point of interest could be implementing persistence of sample data and utilizing the data set for a variety of applications, such as: neural network training, antivirus applications, or dynamically generated *yara* rules.

References

- Andrade, D. (2025) 'PEScan'. Available at: <https://github.com/carett/pescan> (Accessed: 28 April 2025).
- Bugden, W. and Alahmar, A. (2022) 'Rust: the programming language for safety and performance'. arXiv. Available at: <https://doi.org/10.48550/arXiv.2206.05503>.
- Casey, A. (2022) *Performance of serialization libraries in a high performance computing environment*. Available at: <https://hdl.handle.net/10657/13140> (Accessed: 28 April 2025).
- Check Point Software Technologies (2024) *Most common malicious file types received globally view web and e-mail in 2023*, Statista. Statista Inc. Available at: <https://www.statista.com/statistics/1238996/top-malware-by-file-type/> (Accessed: 25 April 2025).
- egmontkob (2025) 'Hyperlinks in terminal emulators'. Available at: <https://gist.github.com/egmontkob/eb114294efbcd5adb1944c9f3cb5feda> (Accessed: 29 April 2025).
- Fulton, K.R. et al. (2021) 'Benefits and drawbacks of adopting a secure programming language: Rust as a case study', in, pp. 597–616. Available at: <https://www.usenix.org/conference/soups2021/presentation/fulton> (Accessed: 27 April 2025).
- mrd0x (no date) *MalAPI.io*. Available at: <https://malapi.io/>.
- Nethercote, N. et al. (2025) 'Valgrind Massif'. Available at: <https://valgrind.org/docs/manual/ms-manual.html> (Accessed: 30 April 2025).
- Proofpoint (2024) *Number of unique threats reported by end users in worldwide organizations in 2023, by threat family*, Statista. Statista Inc. Available at: <https://www.statista.com/statistics/1462339/unique-threats-reported-end-users-by-threat-family/> (Accessed: 25 April 2025).
- Rai, N. and R, V. (2012) 'Windows API based malware detection and framework analysis', *International Journal of Scientific & Engineering Research*, 3(3). Available at: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=8bc641af2d3269d773145cbcb1302cc32d5210e0> (Accessed: 25 April 2025).
- Rust Project Developers (2025) *HashSet in std::collections - Rust*. Available at: <https://doc.rust-lang.org/std/collections/struct.HashSet.html> (Accessed: 29 April 2025).
- Sihwail, R., Omar, K. and Zainol Ariffin, K.A. (2018) 'A survey on malware analysis techniques: static, dynamic, hybrid and memory analysis', *International Journal on Advanced Science, Engineering and Information Technology*, 8(4–2), pp. 1662–1671. Available at: <https://doi.org/10.18517/ijaseit.8.4-2.6827>.
- SonicWall (2024) *Annual number of malware attacks worldwide from 2015 to 2023 (in billions)*, Statista. Statista Inc. Available at: <https://www.statista.com/statistics/873097/malware-attacks-per-year-worldwide/> (Accessed: 25 April 2025).
- Squiblydoo (2024) 'MalAPIReader'. Available at: <https://github.com/Squiblydoo/MalAPIReader> (Accessed: 30 April 2025).
- Statista (2024) *Estimated cost of cybercrime worldwide 2018-2029 (in trillion U.S. dollars)*. Statista Inc. Available at: <https://www.statista.com/forecasts/1280009/cost-cybercrime-worldwide> (Accessed: 25 April 2025).

Appendices

Appendix A – Cargo.toml

```
1. [package]
2. name = "pescan"
3. version = "2.0.1"
4. edition = "2024"
5. authors = [ "caret_" ]
6. description = "static analysis tool for PE files via API import analysis. GPL-3.0-or-later."
7. default-run = "pescan"
8. documentation = "https://carettt.github.io/pescan"
9.
10. [dependencies]
11. anyhow = "1.0.97"
12. clap = { version = "4.5.32", features = [ "derive" ] }
13. camino = "1.1.9"
14. goblin = "0.9.3"
15. scraper = "0.23.1"
16. tokio = { version = "1.44.1", features = [ "full" ] }
17. request = { version = "0.12.15", features = [ "cookies", "http2" ] }
18. dirs = "6.0.0"
19. indicatif = "0.17.11"
20. tabled = { version = "0.18.0", features = [ "ansi" ] }
21. serde = { version = "1.0.219", features = [ "derive" ] }
22. serde_with = "3.12.0"
23. serde_json = "1.0.140"
24. serde_yaml = "0.0.12"
25. rmp-serde = "1.3.0"
26. toml = "0.8.20"
27. csv = "1.3.1"
28.
```

Appendix B – Full Benchmark Code

```
1. use criterion::{black_box, criterion_group, criterion_main, Criterion};
2.
3. use std::process::Command;
4. use std::time::Duration;
5.
6. fn main_process(c: &mut Criterion) {
7.     c.bench_function("main_baseline", |b| {
8.         b.iter(|| {
9.             let process = Command::new("cargo")
10.                .args(["run", "--release", "--",
11.                    "../sample.exe"])
12.                .output().expect("Failed to execute process");
13.
14.             black_box(process);
15.         });
16.     });
17.
18.     let mut all_group = c.benchmark_group("all_details");
19.
20.     all_group.bench_function("txt", |b| {
21.         b.iter(|| {
22.             let process = Command::new("cargo")
23.                .args(["run", "--release", "--", "-A",
24.                    "../sample.exe"])
25.                .output().expect("Failed to execute process");
26.
27.             black_box(process);
28.         });
29.     });
30.
31.     all_group.bench_function("json", |b| {
32.         b.iter(|| {
33.             let process = Command::new("cargo")
34.                .args(["run", "--release", "--",
35.                    "-A", "-f json",
36.                    "../sample.exe"])
37.                .output().expect("Failed to execute process");
38.
39.             black_box(process);
40.         });
41.     });
42.
43.     all_group.bench_function("yaml", |b| {
44.         b.iter(|| {
45.             let process = Command::new("cargo")
46.                .args(["run", "--release", "--",
47.                    "-A", "-f yaml",
48.                    "../sample.exe"])
49.                .output().expect("Failed to execute process");
50.
51.             black_box(process);
52.         });
53.     });
54.
55.     all_group.bench_function("toml", |b| {
56.         b.iter(|| {
57.             let process = Command::new("cargo")
58.                .args(["run", "--release", "--",
59.                    "-A", "-f toml",
60.                    "../sample.exe"])
61.                .output().expect("Failed to execute process");
62.
63.             black_box(process);
64.         });
65.     });
66.
67.     all_group.bench_function("csv", |b| {
68.         b.iter(|| {
69.             let process = Command::new("cargo")
70.                .args(["run", "--release", "--",
71.                    "-A", "-f csv",
72.                    "../sample.exe"])
73.                .output().expect("Failed to execute process");
74.
75.             black_box(process);
```

```
76.     });
77.   });
78.
79.   all_group.finish();
80. }
81.
82. criterion_group!{
83.   name = benches;
84.   config = Criterion::default()
85.     .measurement_time(Duration::from_secs(12));
86.   targets = main_process
87. }
88. criterion_main!(benches);
89.
```

Appendix C – Benchmark Distribution Graphs

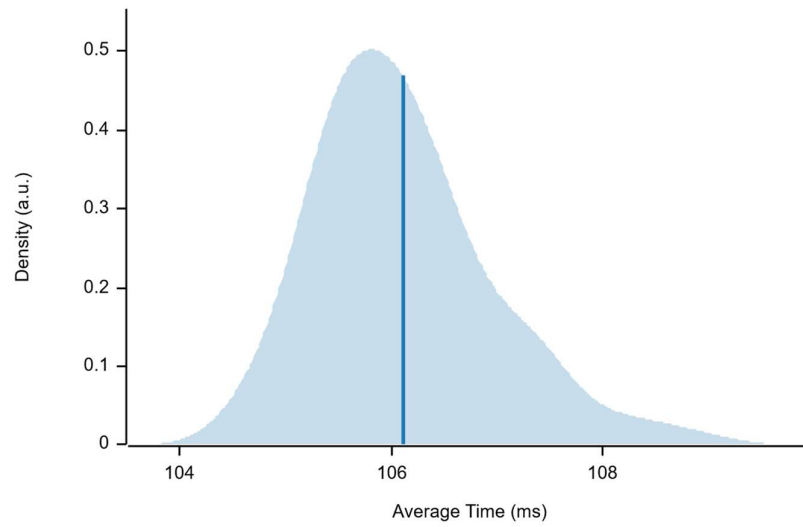


Figure C.1: Distribution of baseline benchmarks

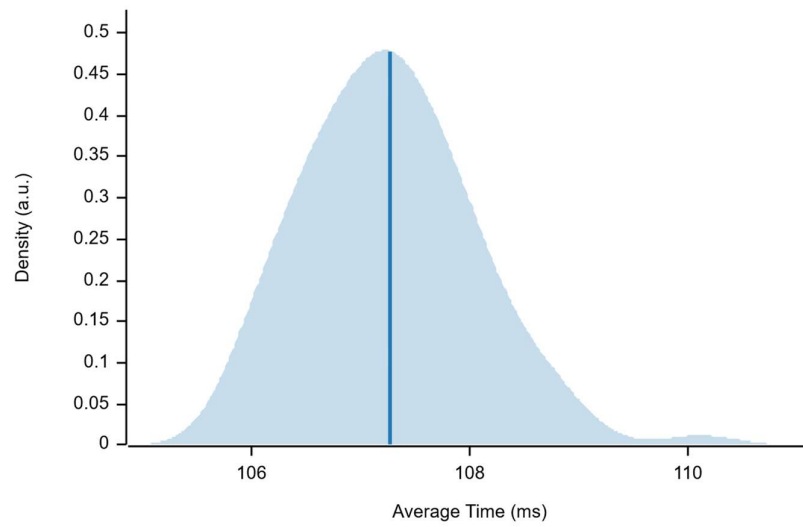


Figure C.2: Distribution of TXT format benchmarks

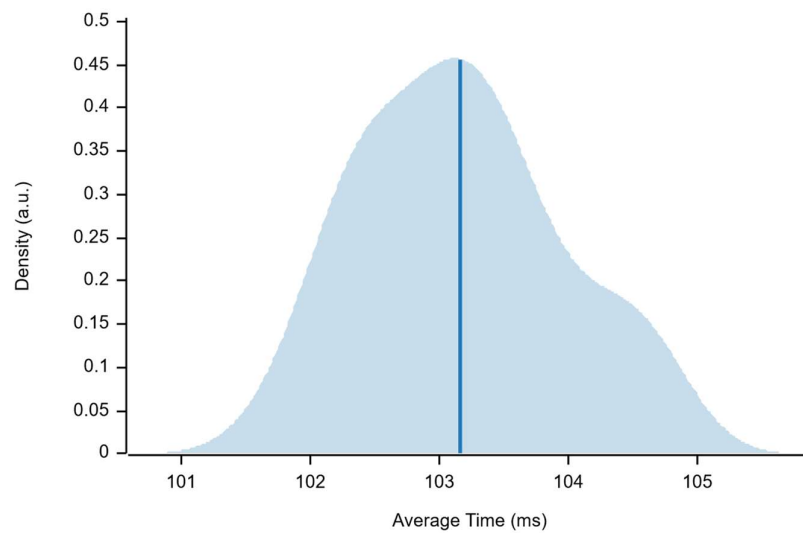


Figure C.3: Distribution of JSON format benchmarks

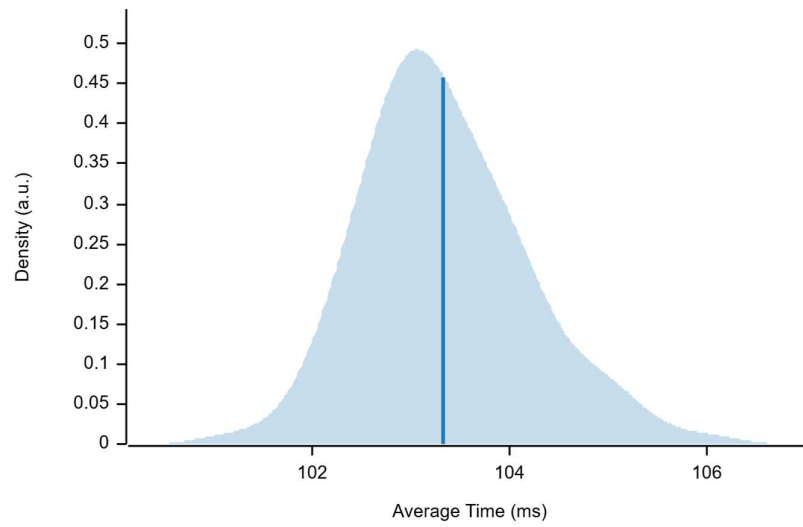


Figure C.4: Distribution of YAML format benchmarks

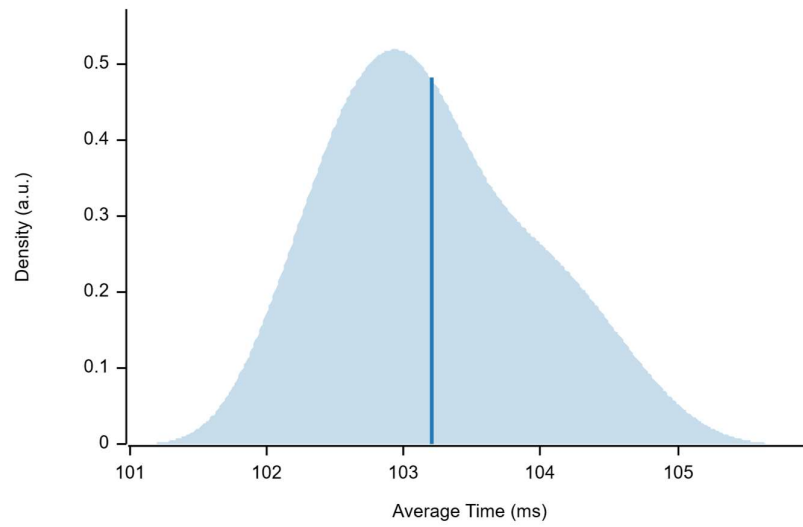


Figure C.5: Distribution of TOML format benchmarks

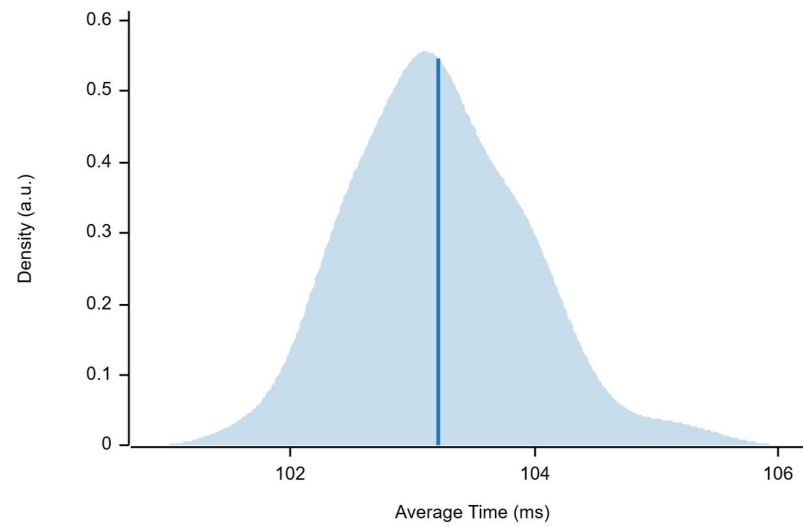


Figure C.6: Distribution of CSV format benchmarks

Appendix D – Full-Size Memory Usage Graphs

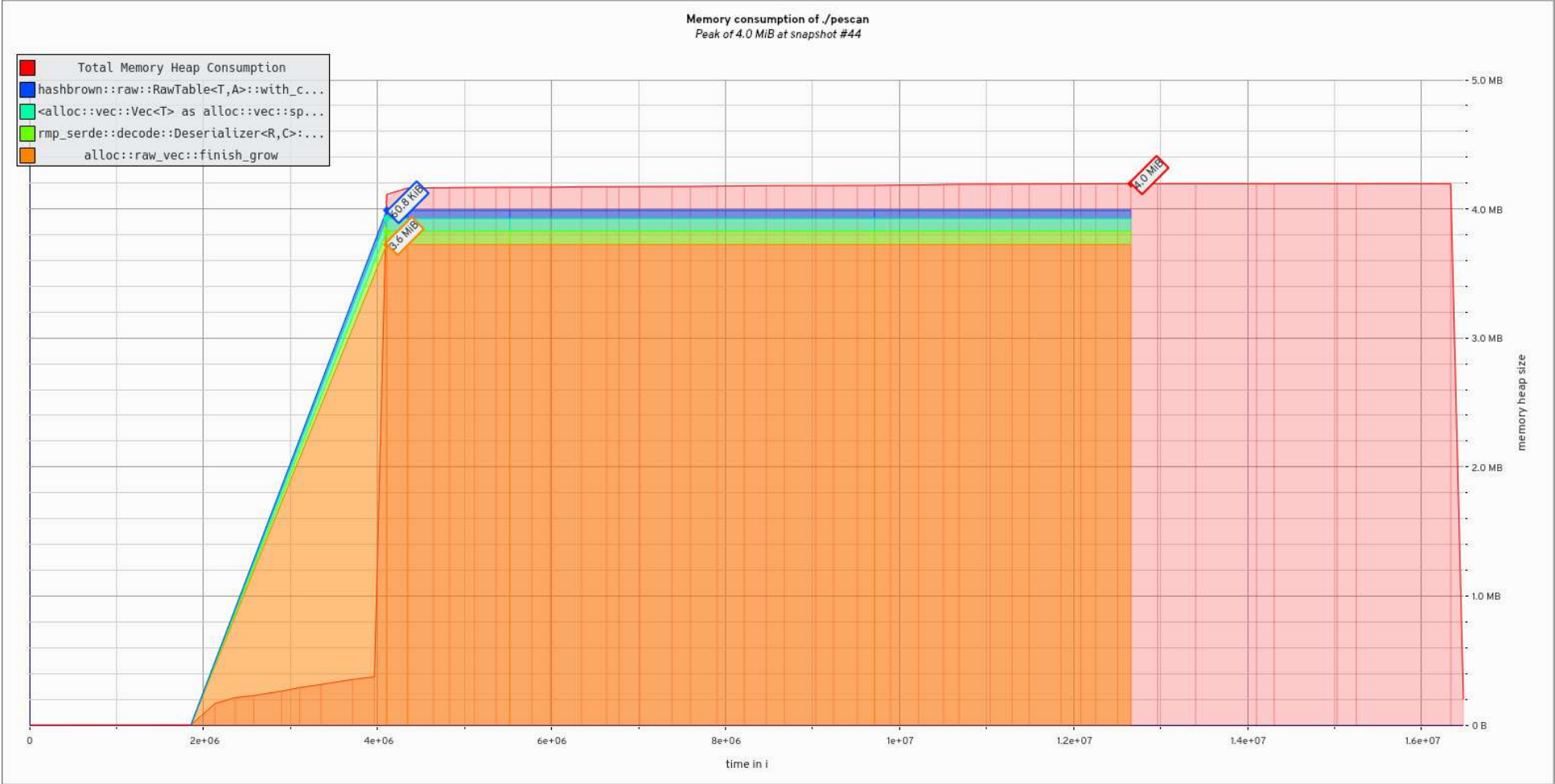


Figure D.1: Full-size memory consumption of PEScan in the TXT format

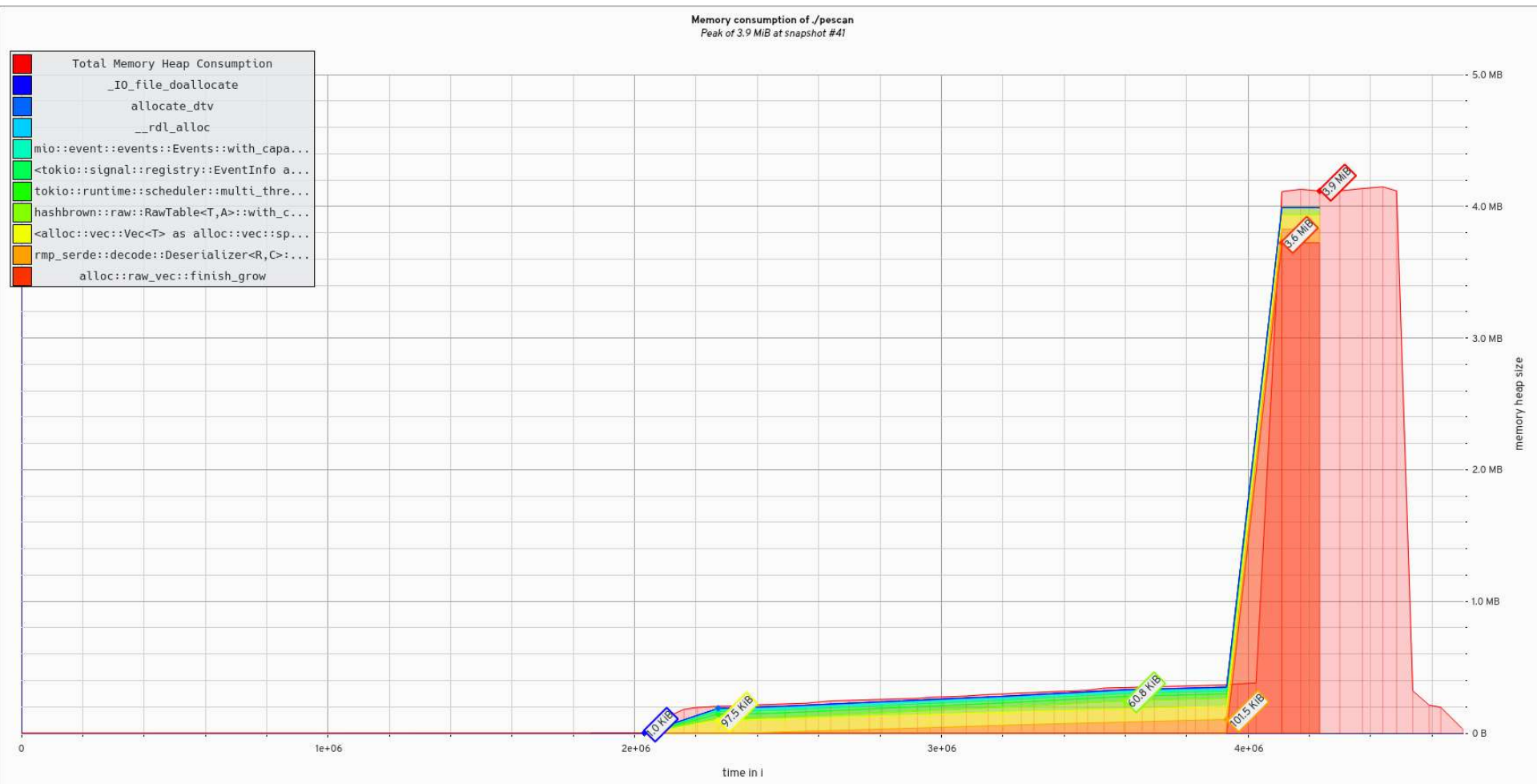


Figure D.2: Full-size memory consumption of PEScan in the JSON format

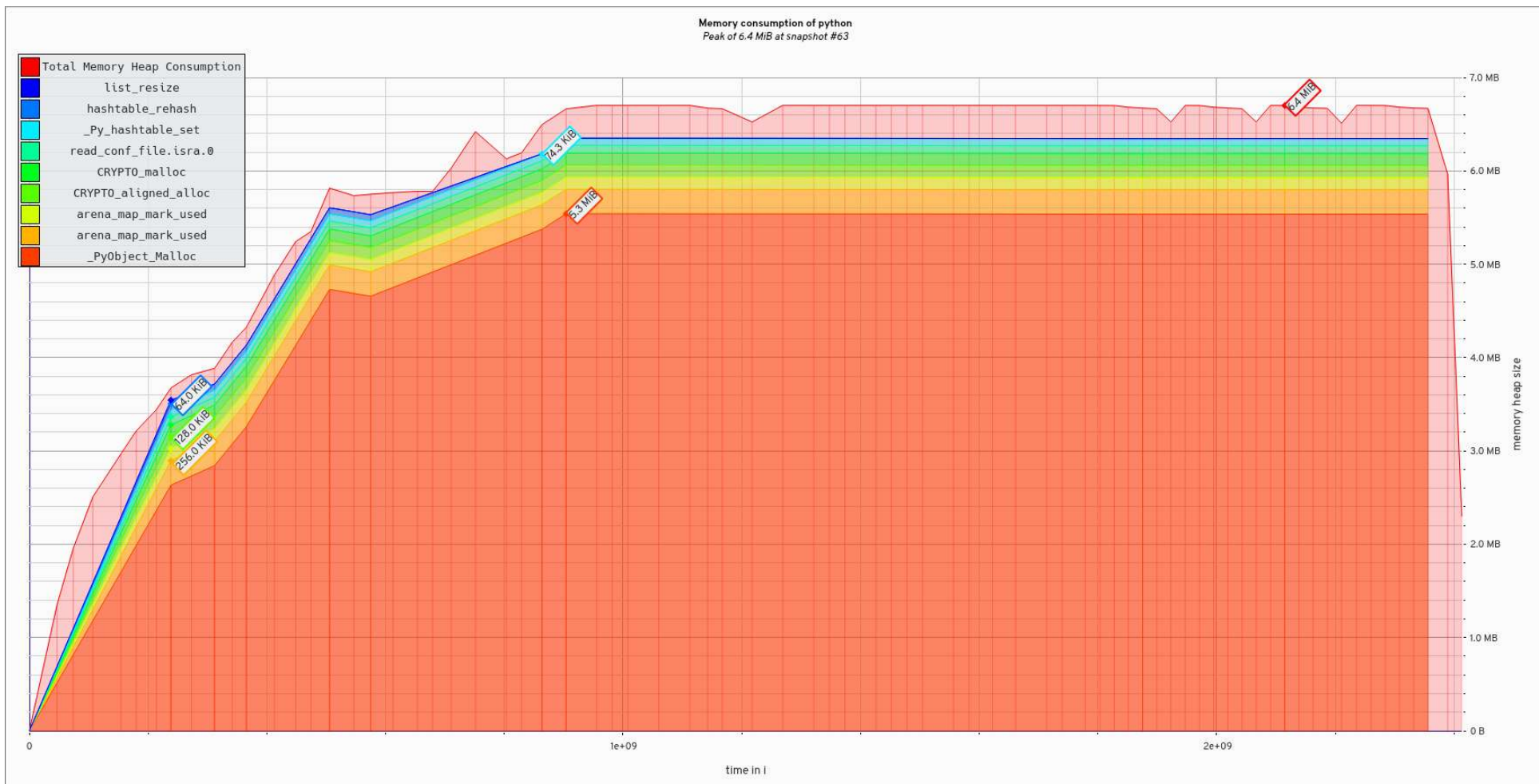


Figure D.3: Full-size memory consumption of MalAPIReader (Squiblydoo, 2024)