UCLouvain

epl

École polytechnique de Louvain

SEMA Evolution: Redefining Malware Analysis Toolchain Architecture

Author: **Manon Oreins**
Supervisor: **Axel Legay**
Readers: **Ramin Sadre, Christophe Crochet**
Academic year 2023–2024
Master [120] in Computer Science

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Axel Legay, for his guidance throughout the year. I also wish to express my deepest gratitude to Christophe Crochet for encouraging me to choose this subject and for his consistent support. I am grateful to the members of the SEMA team—Charles-Henry Bertrand Van Ouytsel, Serena Lucca, and Samy Bettaieb—for helping me understand SEMA and for providing valuable advice and feedback. I would like to thank the members of the jury, Ramin Sadre and Christophe Crochet, for taking the time to review this work.

I am deeply appreciative of my family for their unwavering support and encouragement throughout this process. Specifically, I would like to thank Emily-Yu for supporting me until the end, enduring my stress and mental breakdowns, and providing me with food and love.

Lastly, I extend my thanks to my friends, their encouragement and shared experiences of struggling with our respective master theses made this journey more bearable.

ChatGPT and Phind have been used to paraphrase and restructure this master thesis report.

Abstract

Malware is continually evolving, with their abilities to evade analysis improving daily. This necessitates the development of new tools and techniques to detect and classify the large volumes of malware emerging every day. In this context, the SEMA Toolchain was created as a research project to apply symbolic execution to malware samples, generating a system call dependency graph that serves as a signature for classification.

The aim of this work is to enhance the quality of SEMA, focusing first on maintainability to create a tool that is easier to use and sustain in the future. This involved redesigning the architecture and employing various refactoring techniques. Secondly, the work aims to improve performance by increasing execution speed and reducing memory usage, allowing for faster results and the capability to conduct longer experiments. To achieve these improvements, the PyPy3 Just-in-Time compiler was utilized and memory analyses were conducted to address memory leaks. Performance analysis and testing were carried out to confirm the positive impacts of these modifications.

These enhancements will better prepare SEMA for the rapid evolution of malware and make it more accessible to a broader audience.

Contents

1 Introduction 1

2 Background knowledge 3
- 2.1 Python and Python's interpreters . . . 3
  - 2.1.1 Python . . . 3
  - 2.1.2 The Global Interpreter Lock (GIL) . . . 4
  - 2.1.3 Alternatives to the GIL . . . 5
  - 2.1.4 Pypy's JIT . . . 6
- 2.2 Symbolic Execution . . . 7
  - 2.2.1 Malware analysis . . . 7
  - 2.2.2 Symbolic execution . . . 8
  - 2.2.3 The angr Framework . . . 9
- 2.3 SEMA Toolchain . . . 12
- 2.4 Maintenance and Evolution in Software Systems . . . 13
  - 2.4.1 Maintainability principles . . . 13
  - 2.4.2 Refactoring . . . 14
  - 2.4.3 Types of architecture . . . 15
  - 2.4.4 Virtual machines and containers . . . 16

3 Software issues and implemented solutions 18
- 3.1 Architectural Concerns . . . 18
  - 3.1.1 Monolithic vs micro-services architectures . . . 18
  - 3.1.2 Installation limitations . . . 22
- 3.2 Code Quality Issues . . . 22
  - 3.2.1 Usability . . . 23
  - 3.2.2 Maintainability . . . 25
  - 3.2.3 Other improvements . . . 36
- 3.3 Performance improvements . . . 36
  - 3.3.1 Python performances . . . 37
  - 3.3.2 Refactoring . . . 38
  - 3.3.3 Memory utilization . . . 39

3.4 Testing Methodology 39
3.4.1 Unittest 39
3.4.2 Github CI/CD 41

4 Performance Analysis 42
4.1 Benchmark methodology 42
4.2 Performance comparison 43
4.3 Memory Utilization 48
4.3.1 Before memory improvements 48
4.3.2 After memory improvements 49
4.4 Users Evaluation 50
4.4.1 Installation Process 51
4.4.2 Utilization 51
4.4.3 Adding functionalities 52
4.4.4 Overall Comparison 52

5 Future Work 54

6 Conclusion 57

A Additional graphs 62

B Users evaluation 65
B.1 From Serena Lucca 65
B.1.1 Installation Process 65
B.1.2 Utilization 65
B.1.3 Adding functionalities 65
B.1.4 Overall Comparison 66
B.2 From Samy Bettaieb 66
B.2.1 Installation Process 66
B.2.2 Utilization 66
B.2.3 Adding functionalities 66
B.2.4 Overall Comparison 67
B.3 From Charles-Henry Bertrand Van Ouytsel 67
B.3.1 Installation Process 67
B.3.2 Utilization 67
B.3.3 Adding functionalities 68
B.3.4 Overall Comparison 68
B.4 From Christophe Crochet 68
B.4.1 Installation Process 68
B.4.2 Utilization 69

B.4.3 Adding functionalities 69
B.4.4 Overall Comparison 69

Chapter 1

Introduction

Today, the danger posed by malware threats is greater than ever, with thousands of new samples appearing daily. Various static and dynamic tools are available to detect malware signatures such as Yara [5], allowing to write rules to detect characteristics of binaries statically, and Cuckoo [25], that provides a sandbox where malwares can be executed and observed dynamically.

However, these tools often fail to automatically detect polymorphic malware, which are variants of malware signatures within the same family. Recent works suggests using symbolic execution and machine learning to address these challenges. Unlike traditional execution where actual inputs are used, symbolic execution assumes symbolic values for inputs, allowing the program to be analyzed in terms of these symbols. This approach leads to expressions for variables and outcomes of conditional branches in terms of those symbols, and constraints for the possible outcomes of each branch. By solving these constraints, the possible inputs that trigger a branch can be determined. This methodology is originally used in software testing but is particularly advantageous for scrutinizing malware, as it enables a thorough exploration of the code, leading to the creation of more detailed and informative signatures. [6]

The SEMA (Symbolic Execution for Malware Analysis) toolchain [4] is an open-source tool designed for analyzing, detecting, and classifying malware. It introduces an innovative approach by integrating symbolic execution with machine learning, offering a novel alternative to existing malware analysis tools. The architecture comprises two main components, *SemaSCDG* and *SemaClassifier*. *SemaSCDG* uses symbolic execution to create System Call Dependency Graphs, which are then processed by machine learning models in *SemaClassifier*. SEMA integrates the Python module angr [1], enabling symbolic execution on binaries. The toolchain aspires to become a groundbreaking tool for protecting users against both current and future malware. To achieve this, SEMA requires a redesign of its

monolithic architecture to enhance software maintainability, evolution, flexibility, and scalability. Additionally, the toolchain is becoming difficult to understand and maintain, necessitating a cleaner, more organized code structure through various refactoring techniques. Lastly, SEMA's performance in terms of time and memory is hampered by Python's limitations and memory leaks within the software, which need to be addressed.

As malware rapidly evolves, the demand for a high-performance, maintainable tool has never been greater. Maintainability is a crucial quality attribute that affects the long-term viability and sustainability of software products, influencing both operational efficiency and overall cost of ownership. Maintainability in software refers to how easily a software product can be modified without introducing defects or compromising performance. It includes the ability to adapt the software to changing environments, fix defects, enhance performance, and extend functionality. As software evolves to meet new requirements or leverage emerging technologies, maintainability ensures that these changes can be implemented smoothly and efficiently, minimizing disruption to existing functionality. Regular maintenance activities, such as fixing bugs, improving performance, and adding new features, rely heavily on the software's maintainability. [19]

In this work, significant enhancements are made to SEMA by first transitioning its architecture from a monolithic structure to a microservices model. Then the focus is put on the SCDG part of the toolchain where various refactoring techniques are applied, such as breaking down large classes, reducing the number of required parameters through configuration files, using inheritance, and adding documentation to improve usability, readability, and maintainability. Performance bottlenecks are addressed by optimizing code sections and incorporating the PyPy Just-in-Time compiler to enhance performance. Memory leaks are also identified and eliminated from the toolchain. Performance analysis and user evaluations demonstrate the positive impact of these improvements. The new version of the toolchain is available on a fork of the original repository and can be found here : [33].

This master thesis report is organized as follows. In Chapter 2 background knowledge are provided on Python, symbolic execution, the SEMA toolchain and maintenance and evolution in software systems. In Chapter 3, challenges and constraints within the SEMA toolchain and the implemented solutions are presented. Chapter 4 presents the performance analysis in terms of execution time, memory utilisation and users evaluation. In Chapter 5, potential future works are discussed, followed by Chapter 6 that concludes this work.

Chapter 2

Background knowledge

This chapter serves to furnish the foundational knowledge necessary for understanding the issues and solutions proposed in Chapter 3. Initially, Python and its different interpreters are presented in Section 2.1, going through the main characteristics of each of them. Subsequently, symbolic execution is expounded upon in Section 2.2, accompanied by a succinct overview of static and dynamic Malware analysis. Following this, Section 2.3 provides an exposition of the SEMA toolchain, which serves as the foundation of this document. Lastly, Section 2.4 underscores the significance of maintenance and evolution in software systems, encompassing discussions on architectures and Object-Oriented principles.

2.1 Python and Python's interpreters

2.1.1 Python

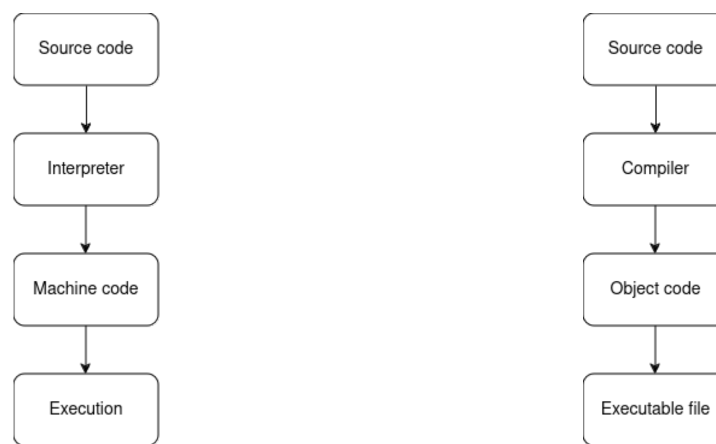
Python [31] is a versatile, high-level programming language designed to enhance the productivity of programmers by its simplicity. It uses dynamic typing and garbage collection that automatize memory management. Python supports multiple programming paradigms such as object-oriented, procedural and functional programming. It also provide a wide range of libraries. Python is primarily classified as an interpreted language, and uses CPython as default interpreter. Understanding the distinction between interpreted and compiled languages, along with the pros and cons of each, provides insight into Python's unique characteristics and its suitability for various programming tasks.

Interpreted languages The execution is a one-step process where the source code is interpreted into machine code that is executed directly at runtime. They tend to be slower because the code is parsed and executed at runtime. It can be

modified while the program is running, which can affect performance but offers flexibility. The interpreter runs on any platform that supports it. Examples : JavaScript, Perl, Python, and BASIC.

Compiled languages The execution is a two-step process from source code to execution—first, the source code is compiled into object code, and then the object code is transformed into an executable file. They offer better performance since the translation to machine code happens before execution, allowing for optimizations that can significantly speed up execution. But compiled code is typically specific to the platform and architecture for which it was compiled, necessitating recompilation for different platforms. Examples: C, C++, C#, and COBOL.

Figure 2.1 illustrates how compiled and interpreted languages process source code.



(a) Key steps of interpretation process

(b) Key steps of compilation process

Figure 2.1: Comparison of compiled and interpreted languages code processing

2.1.2 The Global Interpreter Lock (GIL)

The GIL is a mutex (lock) that prevents multiple native threads from executing Python bytecodes at once in a single process. Introduced in Python 1.5, the GIL ensures that only one thread executes Python bytecodes at a time, preventing race conditions and simplifying memory management. However, it also limits the execution of Python programs to a single processor core, making it unsuitable for CPU-bound tasks that require parallel processing. Figure 2.2 illustrates how the GIL works in python.

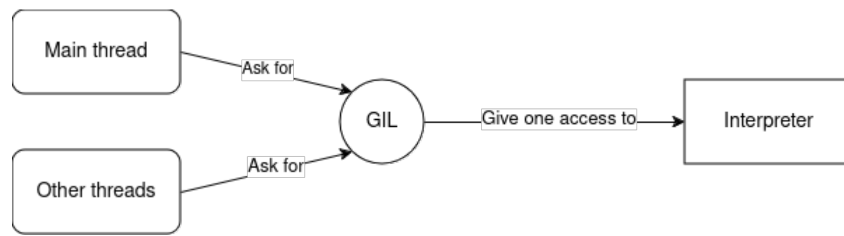


Figure 2.2: Illustration of the GIL in python

Performance implications The GIL's design significantly impacts Python's performance in multi-threaded applications. While it ensures thread safety within the Python runtime, it restricts the ability of Python programs to utilize multiple processors efficiently. This limitation is particularly noticeable in CPU-bound tasks, where the GIL prevents true parallel execution, leading to underutilization of hardware resources.

2.1.3 Alternatives to the GIL

Just in Time Compilation JIT compilers, such as PyPy [44], compile Python code to machine code at runtime, potentially bypassing the GIL's restrictions. PyPy's JIT compiler can execute Python code faster than the standard CPython interpreter, offering a viable alternative for CPU-bound tasks.

Free-Threading An early attempt by Greg Stein to remove the GIL and replace it with fine-grained locking. However, this approach led to significant performance degradation due to the overhead of managing many locks. [15]

Multiprocessing Python's multiprocessing module allows for parallel execution by creating separate processes, each with its own Python interpreter and memory space. This approach circumvents the GIL but introduces overhead due to inter-process communication and synchronization. [36]

GraalVM GraalVM is a universal virtual machine designed to run applications in multiple languages, featuring a Just-In-Time (JIT) compiler that significantly enhances the performance of Python code. Its Python runtime can achieve speeds up to 5-6 times faster than CPython 3.8 for pure Python code, and even more for specific workloads. However, there are some limitations: it does not support subclassing Java classes in Python code, performance for native extensions running as LLVM bitcode is slower, and there may be compatibility issues with some modules or packages. [32]

Jython Jython is an implementation of Python that runs on the Java platform. It allows Python code to call Java methods and libraries seamlessly, enabling Python developers to leverage the vast ecosystem of Java libraries and frameworks. Jython's primary advantage lies in its interoperability with Java, making it particularly useful for integrating Python with existing Java applications or utilizing Java's extensive libraries for data processing, web services, and more. [42]

Numba Numba is a just-in-time compiler for Python that translates a subset of Python and NumPy code into fast machine code. It is designed to optimize numerical computations, making it particularly useful for scientific computing and data analysis tasks. Numba achieves significant speedups by compiling Python code to LLVM bitcode, which can then be optimized and executed at runtime.

Numba can deliver speedups comparable to most compiled languages with minimal effort, developers can simply apply decorators to functions to enable JIT compilation and it supports a wide range of Python constructs, including loops, conditionals, and function definitions, making it flexible for various computational tasks. However, Numba does not support the full feature set of Python and NumPy, which means that some functionalities may not be optimized or may not work at all. [27]

2.1.4 Pypy's JIT

PyPy's Just-In-Time (JIT) compiler operates based on a tracing mechanism. This approach focuses on optimizing the parts of the code that are actually executed, rather than attempting to optimize the entire program upfront. The tracing mechanism allows PyPy to dynamically identify and optimize the most frequently executed sections of the code, which typically includes loops and frequently called functions. When the interpreter encounters a loop or a frequently called function, it traces what happens during the execution of that code block. This trace captures the operations performed by the interpreter, allowing PyPy to analyze and optimize the code path. Once optimized, this trace is compiled to machine code and reused whenever the same code path is encountered again. This process is known as JIT compilation because the optimization and execution happen during the actual execution of the program. [44]

Performance implications By focusing on optimizing the most frequently executed parts of the code, PyPy's JIT compiler can significantly enhance the performance of Python programs, especially those with heavy computational demands. The tracing JIT approach provides flexibility in handling dynamic languages like Python, where the execution flow can change at runtime. This

makes it well-suited for optimizing Python code that involves complex control flows and data structures. The effectiveness of PyPy's JIT compiler depends on the nature of the code being executed. Code paths that are rarely executed or vary significantly in behavior may not benefit as much from JIT compilation. Additionally, the initial startup time might be longer compared to traditional interpreters, although subsequent executions of the same code paths will benefit from the JIT optimizations. [42]

2.2 Symbolic Execution

2.2.1 Malware analysis

Malware analysis is a critical process in cybersecurity, involving the examination of malicious software to comprehend its functionality, behavior, and potential impact. This process aims to neutralize the threat or prevent future attacks. Two primary techniques used in malware analysis are static and dynamic analysis, each with its unique approach and advantages.

Static analysis Static analysis is a method of examining malware focusing on analyzing the code of a binary file to understand its functionality and identify any malicious activity. This technique is used to identify potential security threats in a sample without risking infection of the analysis environment. It serves as an initial step in malware analysis, gathering information about the malware's origin, distribution, and potential indicators of compromise, such as known malicious signatures or suspicious file formats. However, it cannot detect runtime behaviors or malicious actions that occur after the file is executed and it is limited in detecting new or unknown threats since it relies on known patterns and signatures. Yara [5] is an example of a tool doing static analysis.

Dynamic analysis Dynamic analysis, on the other hand, involves executing the malware sample in a controlled environment, such as a sandbox, and observing its behavior interactively. This technique monitors the file system, registry, and network activity of the malware as it runs, providing a deeper understanding of its behavior than static analysis. Dynamic analysis is considered more accurate and comprehensive than static analysis due to its deep behavior analysis capability. This behavior-based detection approach ensures the identification and understanding of new and unknown threats, making it particularly effective against quickly evolving malware or new types of malware that can be difficult to detect using signature-based approaches. [38]

However, this type of analysis is time-consuming and resource-intensive due the need to execute the malware in a virtual environment. Additionally, malwares can detect the presence of a sandbox and alter its behavior to evade detection. Cuckoo [25] is an example of tool doing dynamic analysis in a sandbox.

2.2.2 Symbolic execution

Symbolic execution is a method used in computer science for analyzing a program to understand what inputs cause each part of the program to execute. Unlike traditional execution where actual inputs are used, symbolic execution assumes symbolic values for inputs, allowing the program to be analyzed in terms of these symbols. This approach leads to expressions for variables and outcomes of conditional branches in terms of those symbols, and constraints for the possible outcomes of each branch. By solving these constraints, the possible inputs that trigger a branch can be determined. [6]

This methodology is originally used in software testing but is particularly advantageous for scrutinizing malware, as it enables the exploration of all potential behaviors. It helps mitigate the effects of obfuscation by allowing the analysis of code structures and logic, is less resource-consuming, as there is no need for a sandbox, and can efficiently navigate through complex binaries, including those with self-modifying properties.

Limitation However, this technique can lead to the generation of an exponential number of branches, leading to a significant hurdle known as the "path explosion" problem. Each path necessitates exploration, resulting in an exponential surge in the number of paths to be investigated. This scalability issue impedes the application of symbolic execution to large binaries characterized by numerous branching points. [38] Moreover, designing a good symbolic execution environment is challenging due to the complexity of accurately modeling real-world environments such as memory manipulation and symbolic multi-threading. [3]

To address the path explosion problem, strategic domain-specific optimizations and search heuristics can be applied. These strategies include limiting loop iterations and prioritizing the exploration of certain states, among others. However, while these techniques reduce the search space, the search time remains constrained by the performance of the exploration program. Enhancing the performance of this program can enable faster and deeper exploration, allowing more branches to be examined within the same timeframe.

2.2.3 The angr Framework

angr [7] [45] [29] is an open-source binary analysis framework for Python, designed to facilitate both static and dynamic symbolic ("concolic") analysis using symbolic execution. It provides a suite of tools aimed at solving a variety of tasks related to binary analysis, making it a powerful resource for reverse engineering, vulnerability research, and malware analysis. angr operates by loading a binary into a project, which acts as a container for the binary and its associated analyses. Once loaded, you can configure various settings and load plugins to customize the analysis process according to your needs. Then, angr provides exploration techniques that can be used by the *SimulationManager* to explore the paths of the binary generated by the symbolic execution until a condition is met or the end of the paths. [1]

Modify state options When a new angr project is created, it contains the first SimState, which is an object representing a snapshot of the execution state of the program. Modifying globally state options in angr allows you to tweak the behavior of the execution engine to optimize performance or alter the analysis strategy for specific scenarios. State options are essentially configurations that control various aspects of how angr processes and analyzes binary programs. Some options, like disabling simplification or reducing the level of logging, can help conserve memory, especially in analyses that involve exploring a vast number of states. State options can be modified when creating a new SimState [1]. Listing 1 shows how you can do it:

```
s.options.add(angr.options.LAZY_SOLVES)
```

Listing 1: Add angr option in python

Adding plugins In angr, a state plugin is a component that extends the functionality of a SimState object, allowing for the storage and manipulation of additional data beyond what's provided by the core state attributes. State plugins are designed to hold data and implement interfaces for managing the lifecycle of a state, making them a flexible way to incorporate custom data or behaviors into the analysis process. Implementing a state plugin involves subclassing the *angr.SimStatePlugin* class and overriding methods to define how the plugin interacts with the state. Once defined, a state plugin can be integrated into the analysis workflow by adding it to the state when it's initialized. This can be done manually or by leveraging angr's factory methods to automatically attach plugins to newly created states. [1]

SimProcedures and hooking SimProcedures are Python classes that inherit from *angr.SimProcedure.* They are used to simulate the behavior of functions or procedures within the binary being analyzed. When a SimProcedure is hooked to a function, angr replaces the original function's execution with the SimProcedure's logic. This allows for custom behavior, such as altering the function's output or side effects.

Stashes in angr Stashes in angr are collections of states that are managed by the Simulation Manager. They play a crucial role in organizing and manipulating the state space during symbolic execution, allowing for efficient exploration of a program's behavior. Stashes can be thought of as containers for states, which are snapshots of the program's execution at a particular point in time. The default stash for most operations is the active stash.

When you initialize a new Simulation Manager, the states you create are placed in the active stash. This stash represents the current focus of the analysis, containing states that are actively being explored or evaluated. The active stash is where the majority of the analysis work happens, as it contains the states that are currently being stepped through, filtered, merged, or moved around.

A. Exploration techniques in angr

angr provides a flexible framework for binary analysis, supporting a variety of exploration techniques that allow users to customize the behavior of the analysis process. These techniques can significantly influence how the state space of a program is explored, ranging from altering the search strategy to modifying the stepping process itself. To implement a new exploration technique in angr, you would typically subclass the *angr.ExplorationTechnique* class and override its methods to define the behavior of your technique.

Figure 2.3 illustrates this principle. After implementing your technique, you need to register it with angr so it can be used in simulations. Let's delve into some functions provided by angr for exploration techniques that will be important in the following chapters: `filter`, `step`, `complete`, and `setup`. [1]

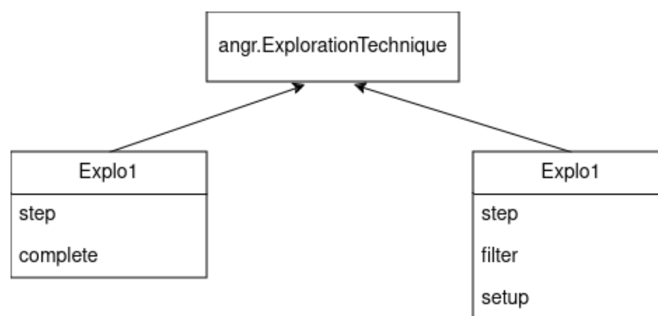


Figure 2.3: Illustration on how *angr.ExplorationTechnique* can be extended

Setup function The setup function is called when an exploration technique is registered with the simulation manager. It is used to initialize the technique, setting up any necessary state or configuration. This function is particularly important for techniques that require initialization logic, such as allocating memory for internal data structures or configuring parameters that will be used throughout the exploration process.

Step function The step function is central to the exploration process. It handles the actual stepping of states through the program, moving from one state to another based on the current state's execution. It iterates through the stash specified by the stash argument, calling `step_state()` on each state and applying the result to the stash list.

Filter function The filter function is a hook that allows exploration techniques to decide which states should be kept or discarded before they are stepped. This is particularly useful for implementing strategies that prioritize certain types of states over others, such as focusing on states that represent interesting conditions or ignoring states that lead to known dead ends. The filter function is called before the step function, giving exploration techniques the opportunity to refine the state space before further exploration proceeds.

Complete function The complete function is a hook that allows exploration techniques to mark a state as completed, meaning it has reached a condition that satisfies the analysis goal. This is useful for techniques that aim to find specific outcomes or conditions within the program's execution. By marking a state as complete, the exploration technique signals that further exploration of this state's descendants is unnecessary, potentially saving computational resources by pruning the state space.

Figure 2.4 illustrates how the exploration process integrates the different functions presented.

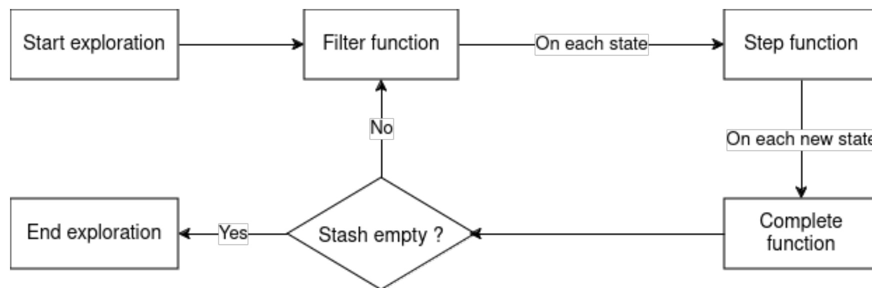


Figure 2.4: Simplified diagram showing the exploration process

2.3 SEMA Toolchain

The SEMA (Symbolic Execution for Malware Analysis) [4] toolchain is an open-source tool designed for analyzing, detecting, and classifying malware. Implemented in Python 3.8, SEMA features a modular architecture to support extensions and adaptability. As shown in Figure 2.5, the architecture comprises two main components: *SemaSCDG* and *SemaClassifier*. Additionally, there is a web application, *SemaWebApp*, which is not depicted in the figure. *SemaWebApp* serves as a graphical interface that allows users to conduct experiments with *SemaSCDG* and *SemaClassifier*. [38] [37] [30]

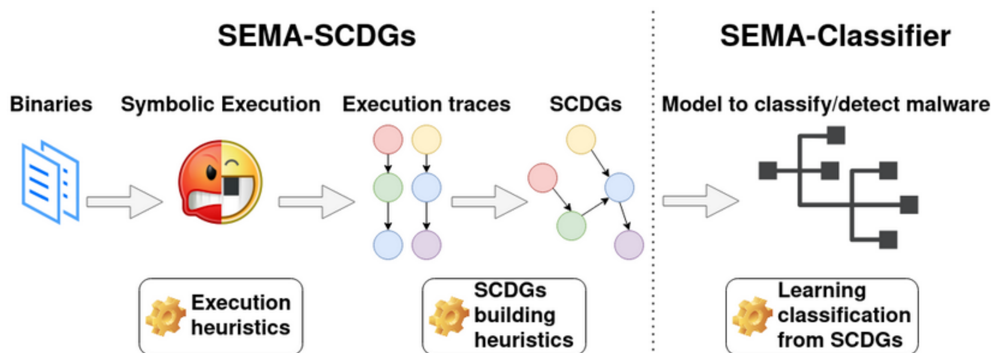


Figure 2.5: SEMA architecture [4]

SemaSCDG *SemaSCDG* uses the angr python framework [1] to perform symbolic execution of binaries during which execution traces are collected to build

System Call Dependency Graphs (SCDG) containing calls and arguments of the that are represented in JSON, GV or GS format. *SemaSCDG* provides different exploration techniques such as DFS (Depth First Search) and CDFS (Coverage Depth First Search) that extend the *angr.ExplorationTechnique* class and implement the `step` and `complete` function of angr. SemaSCDG can take a various number of arguments that influences how the binaries are processed by angr and that control the exploration process.

SemaClassifier *SemaClassifier* is responsible for the malware detection/classification using Machine Learning techniques such as graph mining with GSPAN [46], Support vector machines and deep learning. Taking multiple SCDG in JSON format it can train classifiers and save them for future use. The classifier previously trained can then be loaded and used to classify new unknown binaries.

2.4 Maintenance and Evolution in Software Systems

2.4.1 Maintainability principles

Maintainability in software refers to how easily a software product can be modified without introducing defects or compromising performance. It includes the ability to adapt the software to changing environments, fix defects, enhance performance, and extend functionality.

Maintainability is a crucial quality attribute that affects the long-term viability and sustainability of software products, influencing both operational efficiency and overall cost of ownership. As software evolves to meet new requirements or leverage emerging technologies, maintainability ensures that these changes can be implemented smoothly and efficiently, minimizing disruption to existing functionality.

Regular maintenance activities, such as fixing bugs, improving performance, and adding new features, rely heavily on the software's maintainability. Readability and usability are essential aspects of maintainability, alongside design quality, documentation, and tooling. [24]

Readability Readability refers to how easily and effortlessly a human can read and comprehend the source code of a software program. It involves using clear and consistent syntax, formatting, and naming conventions. Readable code is crucial for maintenance and updates over the software's lifecycle because it facilitates effective collaboration among developers, smooth knowledge transfer, and simplifies the

debugging process. By adhering to best practices in coding styles and conventions such as the ones of OOP [41], developers can create code that is easy to understand, reducing the likelihood of errors and making the software more sustainable over time.

Usability Usability pertains to how effectively end-users can use, learn, or control the system. It encompasses aspects such as the intuitiveness of the user interface, the ease of performing common operations, and the clarity of validation and error messages. From a maintainability perspective, usable software is easier to update and modify because changes made to improve usability are likely to be welcomed by users, reducing resistance to updates and making the software more adaptable to evolving user needs.

Design quality Well-designed software is easier to maintain. Good design principles, such as modularity, separation of concerns, and adherence to design patterns, facilitate modifications and enhancements.

Documentation Comprehensive and accurate documentation, including code comments, architectural diagrams, and user manuals, aids in understanding the software's structure and behavior, making maintenance tasks more straightforward.

Tooling Modern development tools, continuous integration/continuous deployment (CI/CD) pipelines, and monitoring solutions streamline the maintenance process, facilitating rapid detection and resolution of issues.

2.4.2 Refactoring

Refactoring in software development involves restructuring existing code without changing its external behavior. The goal is to enhance the code's internal structure, making it more readable, maintainable, and efficient. This process entails making numerous small changes to simplify the code and reduce complexities, all while preserving the software's functionality.

The code of a software should be habitable, habitable code is a code that makes developers feel at home. It should be easy to read and easy to modify. Software needs to be habitable because it must continuously evolve and adapt. [19]

Why refactor Refactoring enhances code quality by improving the design, structure, and implementation of software, resulting in cleaner and more comprehensible code. Regular refactoring helps teams minimize technical debt, which arises when initial shortcuts lead to poor design decisions that complicate future modifications.

Refactored code is easier to maintain and modify, reducing the risk of introducing new bugs during changes and making the codebase more accessible to new team members.

Additionally, refactoring improves code structure and readability, leading to more modular and reusable components that can be utilized across different parts of the application or in future projects. This clearer, more structured code is also easier to debug and test, enabling developers to identify and resolve bugs more efficiently. [19]

When refactor Refactoring should be performed regularly as part of the software development cycle. It's particularly beneficial when the codebase starts to show signs of becoming hard to understand or maintain also called *bad smells*, or when new features are added that could benefit from a cleaner, more organized code structure. [19]

How refactor Refactoring should be approached methodically, following established best practices. Martin Fowler, a prominent figure in the field, has outlined a catalog of refactorings and described methods to implement them in his book. [19] The process typically involves making incremental changes, testing frequently to ensure that the external behavior of the software remains unchanged, and using version control systems to track changes and revert if necessary.

2.4.3 Types of architecture

The system architecture is crucial in supporting the evolution of software systems. The feasibility and complexity of integrating new functionalities or scaling the system depend on decisions made during the architectural design phase. A well-designed architecture enables the system to evolve, adapt, and remain flexible throughout its lifecycle, ensuring standardized and efficient service delivery.

There are two primary types of system architecture: monolithic architecture, where all services are encapsulated in a single component, and the micro-services approach, where the system is divided into a set of distributed services or components. [43]

Monolith A monolithic architecture refers to a traditional approach where the entire application runs as a single service. All functionalities are contained within a single executable file, and the application's components communicate with each other through well-defined interfaces. Monolithic applications are simpler to develop, test, and deploy because they consist of a single unit. This simplicity makes them ideal for smaller projects or applications with limited complexity.

However, as the application grows, the monolithic architecture can become increasingly difficult to maintain due to its tightly coupled nature. Scaling becomes challenging, and changes to one part of the application can inadvertently affect other parts.

Micro-services Micro-services architecture breaks down the application into a collection of loosely coupled services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. Each service is built around a business capability and deployed independently. Micro-services offer greater flexibility and scalability. They allow for independent deployment, scaling, and updating of services. This architecture is highly suited for agile development environments, enabling rapid iteration and responsiveness to user feedback.

Micro-services introduce complexity in terms of managing inter-service communication, data consistency, and distributed transactions. Additionally, deploying and managing multiple services can be more complex than a single monolithic application. [28]

2.4.4 Virtual machines and containers

Containers and Virtual Machines (VMs) are both popular technologies for isolating and running applications, but they differ significantly in their architecture, performance, and use cases. Understanding these differences is crucial for choosing the right technology for specific needs. [47]

Containers Containers encapsulate an application along with its dependencies into a single object. This ensures that the application runs uniformly regardless of the underlying system. They share the host system's kernel but isolate the application processes and resources. This makes them lighter and faster to start compared to VMs. Containers can run on any system that supports the container runtime, facilitating easier distribution and deployment.

However, while containers isolate processes, they do not isolate the kernel, which could be a concern for certain security requirements. Designed for high portability, scalability, and efficiency, containers are a popular choice for deploying applications across diverse computing environments. These characteristics make Docker a good choice to implement micro-services architecture.

Virtual Machines VMs emulate a complete computer system, including the operating system, on top of a physical or virtualized hardware. This provides a higher level of isolation compared to containers. Each VM runs its own operating system and applications, isolated from other VMs. This is achieved using a

hypervisor, which manages the allocation of resources to each VM. VMs offer strong isolation between applications and the host system, which can be beneficial for security and stability. They can run applications built for different operating systems, making them versatile for legacy applications and they have direct access to the hardware, which can be useful for applications that require low-level access.

However, VMs require more resources than containers because they run a full-fledged operating system, they take longer to start up compared to containers due to the overhead of booting an entire operating system and are less portable than containers because they depend on the specific configuration of the host system.

Figure 2.6 illustrates the differences between Virtual Machines and Containers.

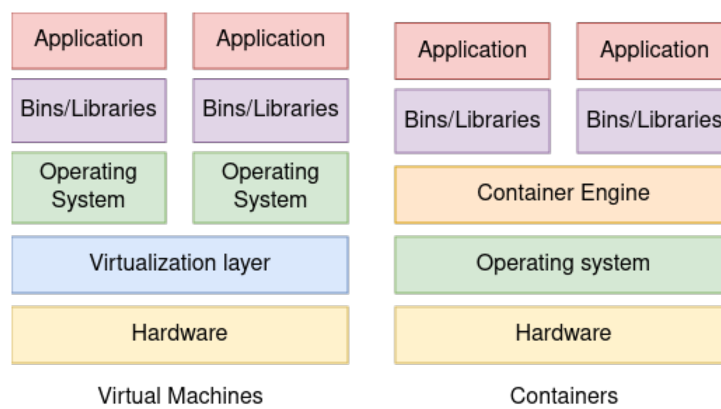


Figure 2.6: Comparison diagram between VMs and Containers

Docker Docker [21] is a platform that uses containerization to package and distribute applications. It simplifies the process of creating, deploying, and running applications by using containers. Docker containers are based on images, which are read-only templates with instructions for creating a container. These images can be shared and reused, making Docker highly efficient for developing, shipping, and running applications.

Chapter 3

Software issues and implemented solutions

In this chapter, we outline the challenges and constraints within the SEMA toolchain across three key domains: architecture design (Section 3.1), code quality (Section 3.2, and performance (Section 3.3). Along with the solutions implemented to tackle down these issues. Following this, in Section 3.4, we elucidate the testing methodology utilized.

3.1 Architectural Concerns

In this Section, the monolithic architecture of SEMA is changed to a micro-services architecture in Section 3.1.1, and installation limitations are presented in Section 3.1.2.

3.1.1 Monolithic vs micro-services architectures

Monolithic architecture issue A monolithic architecture consists of a single, unified codebase that houses all functionalities of the application which fits the architecture of SEMA that is deployed in a single container, even if the *SemaSCDG* and *SemaClassifier* functionalities are pretty well separated. To use the entire toolchain via the web application, the file *SEMA.py* is used to import both *SemaSCDG* and *SemaClassifier* classes, as illustrated in Figure 3.1, and start experiments. It amplifies the phenomenon by linking the two other parts together, forcing a sequential execution while parallel execution could be possible.

For example, if a user wants to analyse a binary with *SemaSCDG* but also train a model with *SemaClassifier*, it is currently impossible to do with the web application of SEMA, as the programs waits for *SemaSCDG* before executing

SemaClassifier. Figure 3.2 illustrates the workflow of the web application.

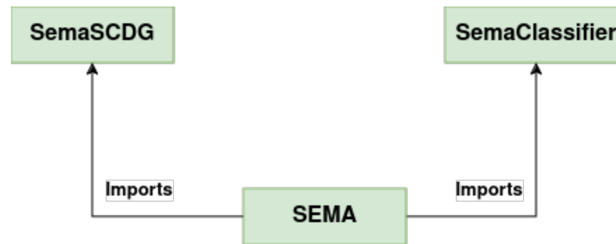


Figure 3.1: SEMA Monolith architecture

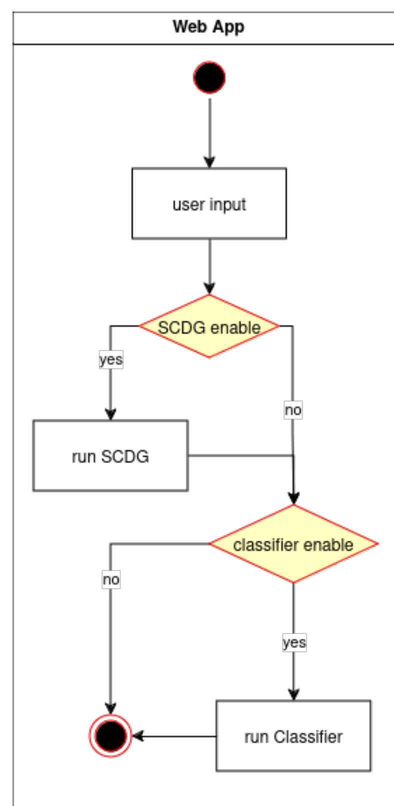


Figure 3.2: Activity diagram of the workflow of *SemaWebApp*

Moreover, as a user can use either the web application or directly *SemaSCDG*, it causes import path issues and a check has to be made on which path has to bee used for the import. Listing 2 illustrates the problem.

```python
try:
    from SemaClassifier.SemaClassifier import SemaClassifier
    from SemaSCDG.SemaSCDG import SemaSCDG
    from helper.ArgumentParserTC import ArgumentParserTC
    from SemaSCDG.clogging.CustomFormatter import CustomFormatter
except:
    from src.SemaClassifier.SemaClassifier import SemaClassifier
    from src.SemaSCDG.SemaSCDG import SemaSCDG
    from src.helper.ArgumentParserTC import ArgumentParserTC
    from src.SemaSCDG.clogging.CustomFormatter import CustomFormatter
```

Listing 2: Imports in `SemaSCDG`

Micro-services solution To address the architectural challenges, the adoption of a micro-services architecture has been considered. As highlighted in [28], transitioning from a monolithic architecture to a micro-services approach offers several advantages. Enhanced maintainability is achieved by decomposing the monolith into independent, self-contained services. This facilitates future developers to implement changes autonomously and enhances code readability due to the reduced complexity of micro-services. Reliability is improved as micro-services, being smaller and offering clear interfaces, are more manageable and seamlessly integrable into larger systems.

To obtain these benefits, the architecture of SEMA has been partitioned into three distinct micro-services : *SemaSCDG*, *SemaClassifier* and *SemaWebApp*. As their name suggest, *SemaSCDG* manages the SCDG aspect of the toolchain, *SemaClassifier* handles the classifier component and *SemaWebApp* provides a user-friendly web interface for easier utilization of the toolchain. *SemaSCDG* and *SemaClassifier* operate independently, enhancing the maintainability of the toolchain and enabling users to conduct parallel experiments on these two parts.

However, *SemaWebApp* relies on the other two components and communicates with them via a concise REST API to retrieve information and start experiments as illustrated in Figure 3.3.

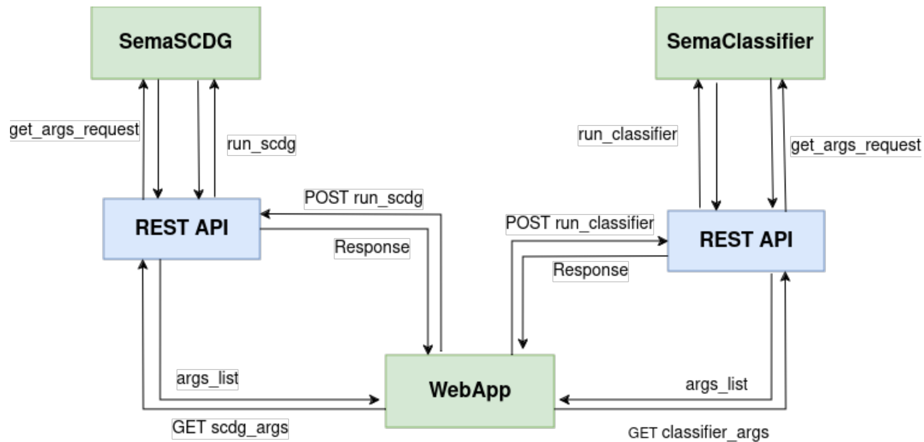


Figure 3.3: SEMA micro-services architecture

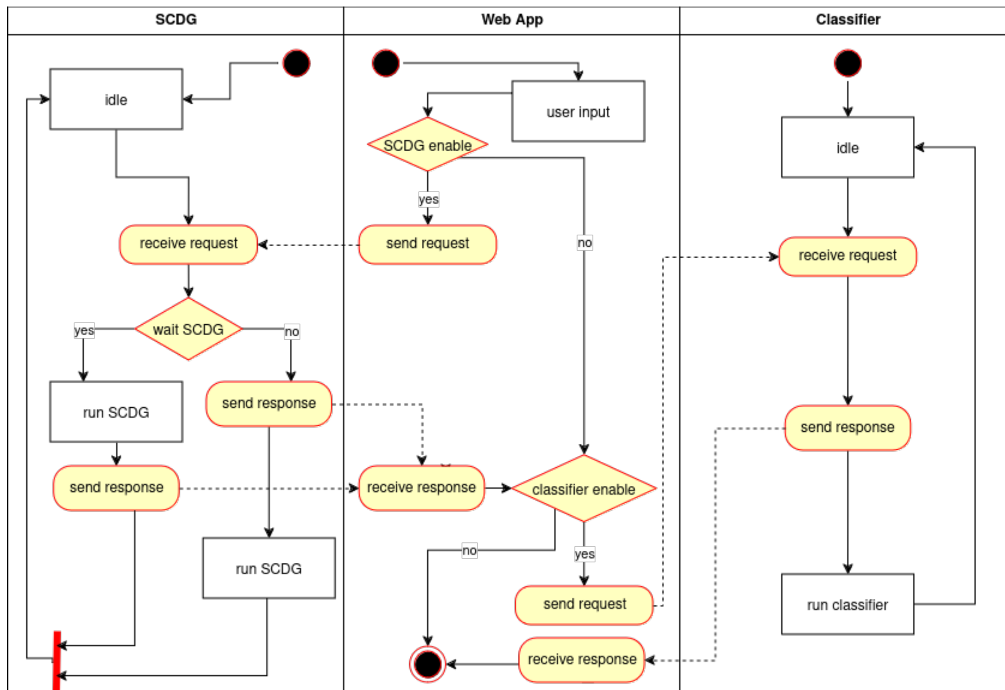


Figure 3.4: Activity diagram representing *SemaWebApp* workflow

Figure 3.4 illustrates the updated workflow when utilizing *SemaWebApp* to execute experiments. In this workflow, *SemaWebApp* initially checks if the SCDG needs to be run. If so, it sends a request containing the user-entered parameters. *SemaSCDG* then verifies if the parameter `wait_SCDG` is set. If it is, indicating that *SemaClassifier* requires the output of *SemaSCDG*, *SemaSCDG* executes first, then responds to *SemaWebApp* that

the request has been received once its execution finishes, thereby causing *SemaClassifier* to wait.

Alternatively, if `wait_SCDG` is not set, *SemaSCDG* promptly responds to *SemaWebApp*, enabling the concurrent execution of *SemaClassifier* in its own container. Upon receiving a request, *SemaClassifier* can immediately responds to *SemaWebApp* and executes its task. This swift response allows *SemaWebApp* to proceed to the next user input without waiting for the execution to finish, facilitating the initiation of new experiments, such as those involving *SemaSCDG*.

In comparison to the initial workflow, this approach enhances the toolchain's performance by enabling the execution of experiments on various configurations as per requirements. This includes utilizing only *SemaSCDG* or *SemaClassifier* individually, running multiple experiments concurrently on both components, or sequentially passing the output of *SemaSCDG* to *SemaClassifier* when needed.

The new micro-services architecture and the communication being based on REST API enable a better segregation of the components, leading to a resolution of the import path problems encountered before.

3.1.2 Installation limitations

Installation limitations Additionally, users faced a limitation where they were compelled to install the entire toolchain even if they only intended to utilize one of these components. This constraint stems from the toolchain's architectural design as a monolithic entity. Operated within a single Docker container, the entirety of the toolchain is installed and executed together, rendering the individual components inseparable. Consequently, users are unable to easily concurrently conduct different experiments on the SCDG and classifier components, diminishing their productivity.

Installation that fits users needs The modularization into micro-services enables users to install only the necessary parts of the toolchain, thereby reducing memory requirements. To streamline installation, a Makefile is provided, offering multiple installation options. The adoption of this new micro-services architecture enhances SEMA's maintainability and usability by introducing a clearer structure and independent components. It also simplifies the process of adding future extensions to the toolchain, expanding its functionalities.

3.2 Code Quality Issues

In this Section, usability issues are addressed in Subsection 3.2.1, including documentation, how to run experiments, the program feedbacks and other improvements that have been made in terms of usability. In Subsection 3.2.2, angr integration, code documentation, dead code, code duplication, arguments in functions, god class, methods size and coupling are discussed. Then, Subsection 3.2.3 introduces the new improvements that have been made.

3.2.1 Usability

A. Documentation

Lack of documentation Another concern relates to the incomplete and ambiguous documentation regarding the installation and utilization of the toolchain. Many aspects require clarification and updating to align more cohesively with the toolchain's usage. Moreover, numerous *TODO* items persist across various *README.md* files, necessitating completion to offer improved support to users.

Adding documentation The documentation has been refined by updating the content of the *README.md* file to include recent additions and by addressing pending tasks outlined in previous iterations (*TODO* items). Additional README files explaining the Makefile and providing more insight on the usage of SEMA have been written, increasing the usability of the toolchain.

B. Running experiments

Long parameter list To utilize *SemaSCDG* with parameters diverging from the defaults, users must input a lengthy array of arguments via the command line.This complexity renders the toolchain more difficult to operate and comprehend. Furthermore, the addition of future arguments would exacerbate the situation, necessitating even longer inputs and diminishing the toolchain's maintainability. An example of user input on Listing 3.

```
python3 SemaSCDG/SemaSCDG.py --CDFS --verbose_scdg --timeout 150
--count_block --csv output/runs/stats/stats_train_prod.csv
--json --limit_pause 5 --simul_state 2 --sim_file
databases/malware-win/train
```

Listing 3: Run SemaSCDG in bash before refactor

Configuration files To address this challenge, configuration files have been introduced, encapsulating all modifiable parameters available in *SemaSCDG*. These files, denoted with the *.ini* extension, are parsed within the code using the Python package *configparser* [16] that is also used in *Django* [14]. A default configuration file, situated within the *SemaSCDG* module in the *config* folder, serves as an illustrative example of values that can be included.

Consequently, users are no longer obligated to specify all parameters via the command line; instead, they need only provide the relative path to the configuration file *SemaSCDG* should reference for execution. This facilitates the creation of custom configuration files or the modification of existing ones, streamlining the launch of runs requiring numerous custom parameters.

Moreover, adding new parameters is simplified, as they only need to be appended to the configuration file to become accessible within the code. Listing 4 is a small example of a configuration file that gives access to four parameters inside the codebase.

```
[SCDG_arg]
log_level_SEMA = INFO
expl_method = CDFS
family = Binaryfamily
plugin_enable = true
```

Listing 4: Small example of a configuration file

angr option flexibility The creation of configuration files has also enabled the modification of angr options that were previously unalterable due to the absence of linked parameters. Now, users can toggle every state option in angr (refer to [1]), granting them control over the exploration of binaries, as elucidated in Section 2.2. This flexibility is advantageous, as these options can influence experiment performance, thus empowering users to adapt them without delving into the codebase.

Plugins flexibility Additionally, configuration files offer the flexibility to enable/disable plugins. Given that certain plugins may prolong execution times, users now have the option to deactivate unnecessary plugins for experiments, thereby conserving time by circumventing redundant operations.

C. Program feedbacks

Terminal submerged by logs During *SemaSCDG* execution, an abundance of logs inundates the terminal, impairing readability. Furthermore, duplicate logs arise from incorrect utilization of the Python *logging* module, exacerbating the clutter and confusion within the terminal interface.

Clearing the logs The bug causing redundant logging entries has been resolved by revising the utilization of the Python *logging* module. Additionally, unnecessary logs have been removed to declutter the terminal interface and improve visibility.

D. Other improvements

Multiple experiments script In order to simplify the utilization of *SemaSCDG*, a shell script named *multiple_experiments.sh* has been developed to facilitate the launch of multiple experiments on *SemaSCDG*. This script enables users to specify the utilization of either Python3 or PyPy3, along with different configuration files for *SemaSCDG* to execute an experiment. Consequently, users can seamlessly launch multiple experiments

within the docker environment without the need to manually initiate the next one upon completion of the previous one.

Experiments configuration Another improvement lies in the improved preservation of experiment configurations, achieved by storing the utilized parameters in a clearer way, reflecting the values passed in the configuration file. Furthermore, each experiment now produces its own *mapping_experiment_name.txt* file, which contains the mapping between syscalls and their corresponding node IDs in the output graph.

Collected data Regarding the saved run data in a *.csv* format, additional columns have been introduced. These columns facilitate distinguishing between the total number of syscalls discovered and the count of distinct syscalls encountered. Similarly, distinctions can now be made between the total number of visited addresses and blocks, providing more granular insights into the experimental results.

Docker compose To streamline the deployment of the toolchain, Docker Compose [20] has been integrated. Each micro-service now includes a *docker-compose.yml* file, encompassing all essential parameters for its initialization. Furthermore, a *docker-compose.deploy.yml* file is placed at the root of the toolchain. Together with the `make run-toolchain` command in the Makefile, this setup facilitates the simultaneous launch and synchronization of all toolchain micro-services, greatly simplifying the deployment of the web application.

PyPi and Dockerhub Another improvement lies in the ability to install the dependencies of the toolchain via PyPi [35]. Additionally, the images of SEMA's Docker containers containing the SCDG part and the Classifier part have been made available on DockerHub [34]. This grants users the convenience of downloading the Docker images directly without the need to build them locally.

3.2.2 Maintainability

A. Angr integration

Poor integration to SEMA Regarding the utilization of *angr* within the toolchain, enhancing its integration into *SemaSCDG* could significantly improve maintainability and correctness.

Currently, within the toolchain, various exploration techniques of *SemaSCDG* extend `SemaExplorer`, which extends itself the `ExplorationTechnique` class of angr and are passed to the angr project for binary analysis. This class provides essential functions such as `step`, `filter`, `complete`, and `setup`, which facilitate managing the exploration process, filtering states, determining exploration completion, and executing setup actions before technique execution, respectively (see Section 2.2).

Presently, while the *complete* function aligns correctly with its counterpart in angr, all exploration techniques within the toolchain utilize the `step` function of angr, assuming responsibility for stash management and state filtering. Every new state generated by the `step` function is put in the *active* stash and then, iterates on this stash to move states on other stashes if necessary.

Additionally, creation of additional stashes and simulation manager attributes at the start of exploration is managed not by `SemaExplorer` but by the main file setting up the angr project, *SemaSCDG.py*.

Adding a `setup` function To enhance the integration of angr with *SemaSCDG*, the management of additional stashes and simulation manager attributes required for binary exploration, previously handled by *SemaSCDG.py*, has been relocated to the `SemaExplorer` class within a new `setup` function. This consolidation centralizes stash logic and leverages the similarly named function provided by angr, thereby enhancing angr's integration within the toolchain. This function is automatically invoked during the creation of exploration techniques, executing all necessary setup operations.

Adding a `filter` function An additional enhancement involves the incorporation of a `filter` function within the `SemaExplorer` class, synchronized with its counterpart in angr. In the initial iteration of the toolchain, the `step` function handled the filtering process by iterating over the *active* stash and manually relocating states to other stashes based on certain conditions.

This process has been refactored into a new `filter` function, responsible for filtering newly generated states from a `step()` call. Upon each state generation, this function determines the appropriate stash for the state by returning the stash's name. By embedding conditions within the `filter` function, states can be directly placed in the correct stash upon generation, bypassing the need to iterate over the *active* stash as previously done when possible.

B. Code documentation and file structure

Code not documented The biggest issue in terms of readability is the lack of docstrings and comments inside the code with most classes and functions remaining unexplained, making the code difficult to understand and hindering future development.

Inconsistent file structure Moreover, the lack of consistency in file structure and naming across different modules hinders the programmer's ability to swiftly locate relevant components within the toolchain. These deficiencies render the structure and the code difficult to comprehend and hampers future development efforts. Figure 3.5 is a segment of the file structure for SEMA, serving as an illustrative example. It reveals numerous directories labeled as *output*, complicating identification of their specific purposes.

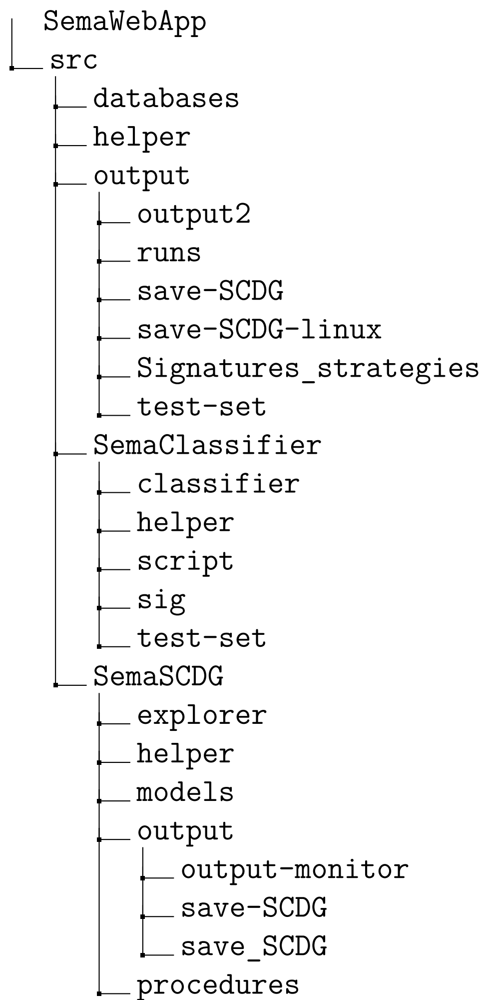


Figure 3.5: File structure before refactoring

Improving documentation and file structure Additionally, docstrings have been integrated into the codebase to enhance readability and facilitate comprehension for future developers. File structures and function names have been revised to better align with their respective functionalities, thereby enhancing readability. Moreover, each module within the toolchain now adheres to a standardized base structure, as showed in Figure 3.6, for consistency and ease of navigation.

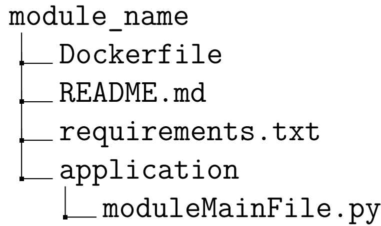


Figure 3.6: Standardized base structure

C. Dead code

Presence of dead code Dead code encompasses instances where code has been commented out but never reused, as well as conditional statements where the conditions are never met, rendering the enclosed code inactive. Dead code adds unnecessary weight to the codebase and can potentially mislead programmers, who may assume that certain instructions are executed when, in reality, they are not. Commented-out code further diminishes code readability by needlessly increasing the volume of code.

Removing dead code Removing dead code is straightforward, as the only solution is to simply eliminate it. Commented-out code has been deleted, enhancing clarity and readability. Conditional statements with branches that never execute have been reviewed to determine whether the code should execute at some point, leading to a modification of the condition if necessary, or if it can be removed from the codebase altogether.

D. Code duplication

Presence of code duplication The toolchain presents snippets of code that are very similar or identical at multiple different places in the code. Code duplication is generally undesirable, as it's preferable to have a unified and concise codebase. Additionally, duplicated code contributes to increased codebase size and complexity, making it more challenging for programmers to read and comprehend. As a tangible example, consider the following code fragment extracted from the file *CustomSimProcedure.py* in Listing 5, which recurs in multiple sections of the file.

```
if hasattr(string, "decode"):
    args[index_str] = string.decode("utf-8")
else:
    args[index_str] = string
```

Listing 5: Duplicated code

Another instance of code duplication within the toolchain is characterized by the presence of conditional statements where the same instruction is executed in both branches. This

redundancy is exemplified within the `CustomSimProcedure` class. Such duplication not only increases the verbosity of the code but also hampers its readability and maintainability. Eliminating such redundant conditional statements can lead to a more streamlined and efficient codebase. Listing 6 illustrates the issue.

```
if name in self.FASTCALL_EXCEPTION:
    project.hook(
            dic_symbols[name],
            simprocedure(
                cc=SimCCMicrosoftAMD64(project.arch)
            ),
        )
else:
    project.hook(
            dic_symbols[name],
            simprocedure(
                cc=SimCCMicrosoftAMD64(project.arch)
            ),
        )
```

Listing 6: Redundant conditional statements

Removing code duplication To resolve the code duplication problem within the codebase, the *Extract method* and *Rename method* strategies [19] have been used. This strategy involves isolating duplicated code and relocating it into a new method with appropriate parameters. The newly created method should possess a clear and descriptive name, enabling future programmers to grasp its functionality without delving into the implementation details.

Applying this technique to the example from Section 3.2 yields the following code snippet in Listing 7. Now, all instances where this code was duplicated have been replaced by a call to this function, thereby enhancing clarity and eliminating code redundancy.

```
def __decode_string(self, string):
    if hasattr(string, "decode"):
        return string.decode("utf-8")
    else:
        return string
```

Listing 7: Function from duplicated code

To streamline conditional statements where both branches executed identical instructions,

the strategy employed involved simplifying the statements by eliminating the `else` condition. Instead, only one set of instructions from the branches was retained, thereby removing redundancy and potential confusion while enhancing readability.

E. Arguments in functions

Lots of arguments in functions Another sign of poor code quality in the toolchain is the utilization of objects with extensive parameter lists, which is suboptimal as it may involve unnecessary parameters and complicates code comprehension and maintenance. An illustration of this can be found in the `SemaExplorer` class shown in Listing 8, where the constructor necessitates 22 parameters. Such a lengthy parameter list not only increases the likelihood of including superfluous parameters but also exacerbates the difficulty in understanding and maintaining the codebase.

```
def __init__(self, simgr, max_length, exp_dir,
    nameFileShort,scdg,call_sim,eval_time=False,
    timeout=600,max_end_state=600,max_step=100000000000000000000,
    timeout_tab=[1200,2400,3600],jump_it=100000000000,
    loop_counter_concrete=10000000000,jump_dict={},
    jump_concrete_dict={}, max_simul_state=1,
    max_in_pause_stach=500,print_on=False,
    print_sm_step=False,print_syscall=False,
    debug_error=False,memory_limit=True)
```

Listing 8: Constructor of `SemaExplorer` before refactor

As this class acts as a foundational class for subclasses that implement exploration techniques like `SemaExplorerDFS`, all these parameters must also be provided in the constructor of these subclasses. In the current version of the toolchain, this is accomplished by passing an object to the constructor of the subclass. Subsequently, the subclass transfers the attributes of this object to the `SemaExplorer` class. Listing 9 is a snippet exemplifying a portion of the constructor of `SemaExplorerDFS`.

```python
def __init__(self,simgr,max_length,exp_dir,nameFileShort,worker):
    super(SemaExplorerDFS, self).__init__(
        simgr,max_length,exp_dir,nameFileShort,
        worker.scdg,worker.call_sim,worker.eval_time,worker.timeout,
        worker.max_end_state,worker.max_step,worker.timeout_tab,
        worker.jump_it,worker.loop_counter_concrete,worker.jump_dict,
        worker.jump_concrete_dict,worker.max_simul_state,
        worker.max_in_pause_stach,worker.print_on,
        worker.print_sm_step,worker.print_syscall,worker.debug_error
    )
```

Listing 9: Constructor of `SemaExplorerDFS` before refactor

Removing long parameter list To reduce the extensive list of parameters required by certain objects and functions in the codebase, the implementation of configuration files (refer to Section 3.2.1) has been highly beneficial. Many parameters previously required by classes and functions can now be directly accessed through the configuration file fields of the experiment. Listing 10 is the updated constructor for `SemaExplorer`.

```python
def __init__(self,simgr,exp_dir,nameFileShort,scdg_graph,call_sim)
```

Listing 10: Constructor of `SemaExplorerDFS` after refactor

All optional parameters have been removed, leaving only those not present in the configuration files and the `scdg_graph` parameter, which is the structure where the SCDG will be constructed. The constructor of `SemaExplorerDFS` now demonstrates a simplified parameter list that is shorter and more readable. Eliminating long parameter lists enhances the program's understandability for future developers and improves long-term maintainability.

F. God class

God class in SEMA The presence of a very large class, commonly referred to as a "god class" [19] indicates a lack of maintainability in the codebase. Specifically, the file *SemaSCDG.py* contains the `SemaSCDG` class, spanning a substantial 1500 lines of code, which, within the context of SEMA, is considered excessive. This class is tasked with a multitude of responsibilities, including creating the angr project for binary exploration, setting up plugins, initiating exploration, managing binary packing, configuring stashes required by the simulation manager, collecting execution data, and constructing the SCDG by processing stash contents post-exploration. Such a wide array of responsibilities consolidated within a single class leads to ambiguity regarding the abstractions it provides and renders the program more challenging for developers to comprehend.

Taking down god class To remove the status of god class from `SemaSCDG`, the first step involved utilizing the *Extract class* strategy [19]. This strategy entails dividing the class's responsibilities and assigning them to new classes specifically designed to handle them. By consolidating different functionalities into smaller, specialized classes, this approach aims to enhance code comprehension.

In the case of `SemaSCDG`, a new class named `DataManager` was created to segregate its functionality. `DataManager` is responsible for gathering data from different runs and managing the creation and storage of CSV files. Subsequently, the *Move method* technique [19] was employed. This technique involves relocating methods that are not closely related to the class they reside in, into classes where they are more fittingly utilized.

For instance, the `setup_stash` method, which initializes various global states of the stashes used by the simulation manager, was moved into the `SemaExplorer` class, where logic pertaining to stash management is consolidated. Additionally, further subsections will contribute to reducing the size and complexity of the `SemaSCDG` class by applying additional refactoring techniques.

Another large class that has seen its size reduced is the `CustomSimProcedure` class using the *Extract class* strategy. Previously, this class was burdened with handling both symbolic procedures and constructing the system call dependency graph (SCDG). To address this, the logic pertaining to the SCDG was extracted from the class and relocated into a new class named `SyscallToSCDG`. This separation effectively reduced the size of `CustomSimProcedure`, enhancing its maintainability and readability.

G. Methods size

Long methods In addition to being a god class, the `SemaSCDG` class exhibits the presence of lengthy methods, contributing to making the code less habitable to decreased code readability and maintainability [19] as modifications become harder to do. Despite encompassing numerous responsibilities, as outlined in the subsection "Large class," the class diagram depicted in Figure 3.7 reveals a relatively small number of methods. This suggests that to occupy its 1500 lines of code, the methods are likely either excessively lengthy or dominated by a single substantial method with the rest being smaller.

In this case, one method stands out as particularly large: `build_scdg`, spanning 1000 lines. Additionally, `build_scdg_fin` and `start_scdg` exceed 100 lines each, constituting substantial portions of code. Meanwhile, the remaining methods are comparatively smaller. Similar issues are observed in other classes such as `CustomSimProcedure` and `GraphBuilder`, where long methods necessitate refactoring to enhance code quality.

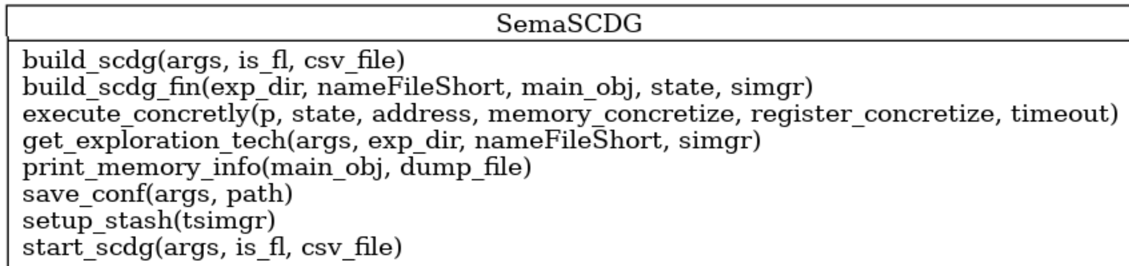


Figure 3.7: Class diagram of *SemaSCDG* class of the initial version of the toolchain with only the methods

Splitting long methods To reduce the size of lengthy methods, which can make the code less habitable, the *Extract Method* and *Rename Method* techniques have been employed [19]. The *Extract Method* involves identifying pieces of code that fit nicely together within a method and extracting them into a new method, thereby reducing the size of the original method and enhancing its readability and comprehensibility. The *Rename Method* involves naming the new method in a way that clearly describes its functionality, eliminating the need for comments to explain what the method does.

However, this process should be used judiciously, as excessive splitting can result in numerous small methods, requiring programmers to frequently switch contexts to understand sub-procedures.

Referring back to the example of `SemaSCDG`, the `build_scdg` method has been divided into several shorter methods, such as `get_binary_args` and `get_entry_addr`, among others. This refactoring reduced the class size to 750 lines of code, which is a significant improvement, halving the original length. Figure 3.8 displays the class diagram of `SemaSCDG` post-refactoring, illustrating an increase in the number of methods compared to before. The class having a reduced size, it indicates that the code is now more modular, with functionalities distributed across multiple methods rather than concentrated in a single method.

Furthermore, the class does not contain an excessive number of methods, demonstrating that the refactoring did not result in excessive splitting and that the new methods are appropriately sized. The same techniques have been applied in other places in the codebase, like in *CustomSimProcedure.py* and *GraphBuilder.py*

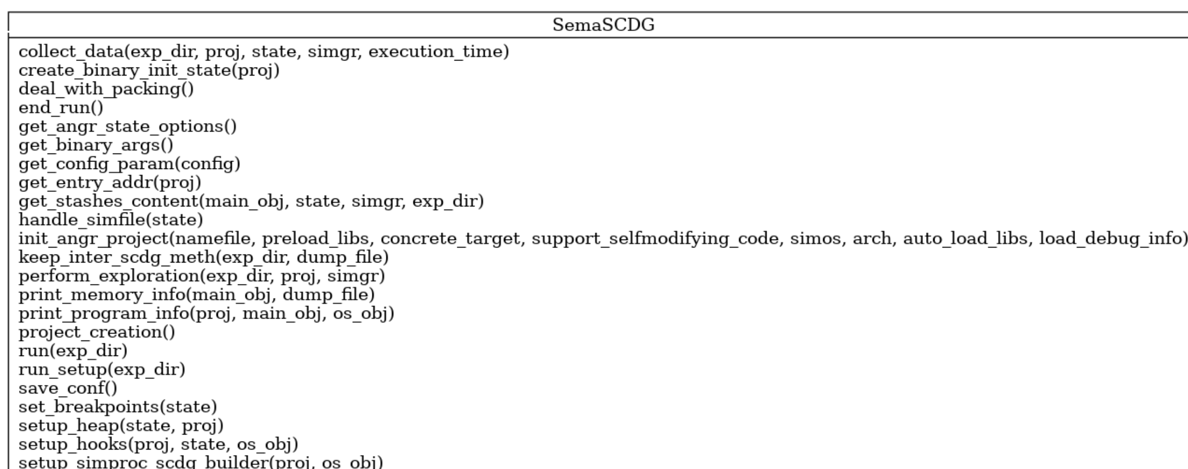


Figure 3.8: Class diagram of the class `SemaSCDG` of the refactored version with only the methods

H. Coupling

High coupling Coupling is the degree to which different software components depend on each other. One bad smell related to that is tight coupling [19]. Classes should not be connected to other classes extensively, the better is to try to keep short the number of classes linked to a class. Indeed, it ensures a better separation between the components and such, a better encapsulation and abstraction leading to a good quality code.

The class `SemaSCDG` possesses this bad smell as we can observe it through the number of classes that are imported and used by the class. The codebase contains a significant number of imports originating from the same directories, such as *plugin* and *explorer*.

Notably, in the case of plugins, `SemaSCDG` imports not only specific components but the entire content of the file. This practice carries inherent risks, as it may lead to name conflicts between imported functions.

Moreover, all classes related to exploration techniques are imported, even though `SemaSCDG` only requires the ability to instantiate the one corresponding to the received parameter.

Consequently, `SemaSCDG` gains access to an excessive amount of data from other classes, despite necessitating only small portions of them. This overabundance of imported data increases coupling and diminishes the maintainability and quality of the code [19].

Figure 3.9 illustrates the plugins classes imported by `SemaSCDG`.

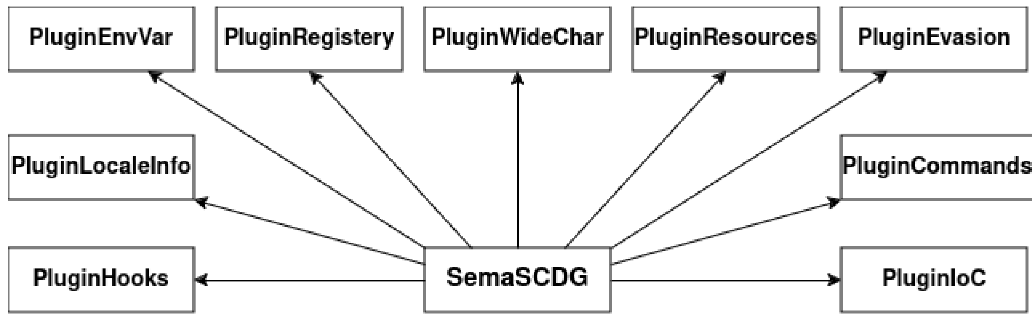


Figure 3.9: Diagram showing the usage relation between SemaSCDG and the plugins before refactor

Aiming for loose coupling Aiming for loose coupling involves minimizing the dependencies a class has on other classes, thereby increasing encapsulation and clarifying the abstractions the class provides. To achieve loose coupling in the `SemaSCDG` class, delegation has been utilized. Given that many imports originate from the same packages, such as *explorer* and *plugin*, a new class has been created within these packages to provide specific functions for interacting with the various classes in the package. This approach reduces the number of elements `SemaSCDG` and other classes need to access when using the package.

For instance, in the *plugin* package, Figure 3.10 illustrates the interaction between `SemaSCDG` and the new class, `PluginManager`, which itself interacts with the other members of the package.

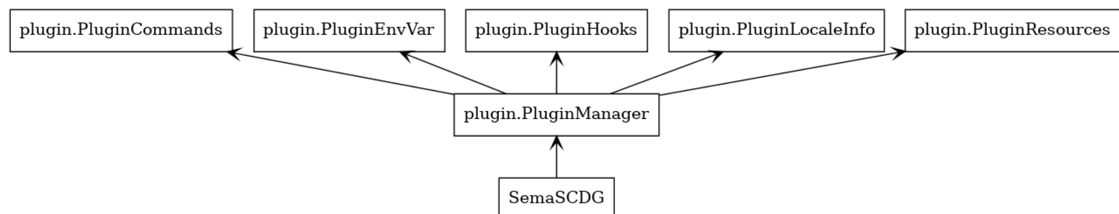


Figure 3.10: Diagram showing the usage relation between the classes related to plugins and *SemaSCDG* after refactor

Applying the same procedure to the *explorer* package, a new class named `SemaExplorerManager` has been created to fulfill the same role as `PluginManager`. This reduces the number of imports required by `SemaSCDG`. This approach enhances data encapsulation for other classes and ensures *SemaSCDG* maintains loose coupling, thereby improving software quality [19].

3.2.3 Other improvements

Splitting Linux and Windows logic Another refactoring aimed at improving the code quality of the SCDG part of the toolchain involved separating the Linux and Windows logic in the *CustomSimProcedure.py* file.

Initially, both logic were intertwined, leading to the execution of unnecessary code and causing confusion for programmers who struggled to distinguish which parts were necessary for each operating system. To separate the two logic, inheritance was used. The `CustomSimProcedure` class now serves as the base class for two new subclasses, `WindowsSimProcedure` and `LinuxSimProcedure`. Common methods that are OS-independent remain in the base class, while OS-specific functions are placed in their respective subclasses. `CustomSimProcedure` now includes abstract methods that must be implemented by the subclasses, enabling polymorphism. The two subclasses implement these methods according to their specific behaviors.

Figure 3.11 illustrates this process by showing the relationship between `CustomSimProcedure`, `LinuxSimProcedure`, and `WindowsSimProcedure`. This refactoring has increased the program's maintainability by clearly separating different logic, enhancing readability, and making it easier to add support for additional operating systems without interfering with existing ones.

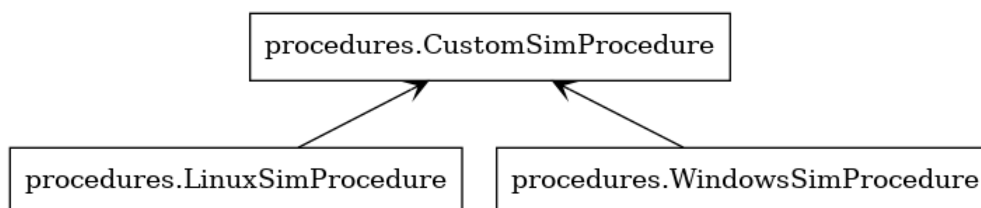


Figure 3.11: Class diagram of the package *procedures* of the refactored version

Simplification of conditional statements The last improvement made in various parts of the code is the simplification of conditional statements. The initial version of the toolchain frequently contained multiple nested conditional statements and conditions that manually checked numerous values separated by AND operators. To simplify and streamline the code, nested conditions were consolidated into single conditions, and multiple values were placed into lists for the condition to check membership.

These changes make the code clearer and easier for programmers to read, thereby enhancing productivity in the future as less time will be spent understanding the codebase.

3.3 Performance improvements

In this Section, different aspects of improving performances are considered. First, in Subsection 3.3.1, Python performance in SEMA is discussed. Second, in Subsection 3.3.2,

how the performances have been enhanced due to refactoring is introduced. Finally, in Subsection 3.3.3, the memory utilization of SEMA and how to reduce it is presented.

3.3.1 Python performances

Bottleneck As discussed in Section 2.1, employing Python with the CPython interpreter exposes certain performance bottlenecks. Given Python's interpreted nature, it tends to run less efficiently on prolonged applications due to its inability to optimize code execution.

Another constraint of Python arises from the Global Interpreter Lock (GIL), which permits only one thread to use the Python interpreter simultaneously. Currently, the toolchain exclusively relies on CPython for conducting experiments, leading to suboptimal performance and prolonged execution times. This, in turn, hampers user productivity.

Using PyPy To address the performance bottleneck, an investigation into utilizing PyPy3 [44], a just-in-time compiler, was undertaken. The angr documentation [1] hinted at a potential speedup of up to 10 times. Indeed, as proposed in [11], PyPy3 has demonstrated the capability to enhance the performance of Python programs over time, albeit with potentially longer startup phases as it generates machine code and loads program libraries.

However, once the code is optimized, PyPy3 has the potential to outperform CPython. It is important to note that PyPy3 requires memory to store the generated machine code, leading to increased memory requirements for the program. Given that, the SEMA toolchain is designed to conduct experiments on numerous binaries, each undergoing repetitive processing, utilizing PyPy3 could significantly enhance performance.

To evaluate the impact of using PyPy3 on the program, benchmarking was conducted to compare performance using CPython and PyPy3. The SCDG's container provides both Python3 and PyPy3 installations, offering users the flexibility to choose the preferred option.

Parameters tuning Another approach to enhancing the performance of the toolchain involves adjusting various parameters provided by both the SCDG and the angr framework [1]. The angr framework offers a plethora of state options that can be toggled, and users can modify these settings along with other parameters such as the number of states explored simultaneously, the allowed number of jumps, and the maximum number of iterations within a loop, among others, through configuration files. These parameters significantly influence binary analysis and can affect the duration and depth of exploration.

Parameter tuning is tailored to each individual binary, and while the toolchain does not provide an automated tuning tool, users have the flexibility to adjust parameters directly and even create scripts for parameter tuning utilizing the configuration files.

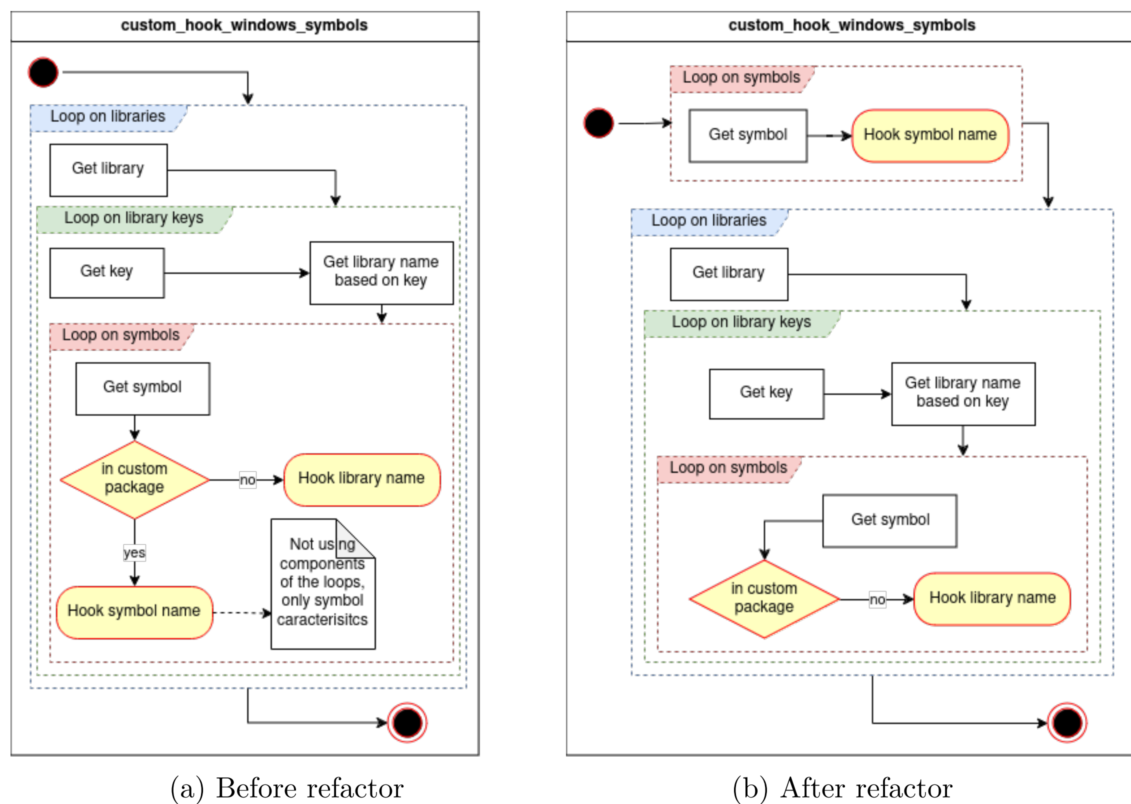
3.3.2 Refactoring

To enhance even more performances, attention was directed towards improving execution time through refactoring, resulting in a significant performance boost for the toolchain. Notably, the most substantial improvement was observed in the reduction of hooking time before exploring each binary, which has been slashed to less than 5 seconds for the majority of analyzed binaries.

Figure 3.12 illustrates the refactor that has been applied to the function `custom_hook_windows_symbol` resulting in the previously described improvement. Prior to the refactor, the function looped through the symbols found in the binary and hooked the symbol name if it was present in the `custom_package` dictionary.

However, it did not utilize any variables from the two outer loops, resulting in the repetitive execution of this hooking process at each iteration unnecessarily. The refactor involved extracting this repetitive process into a separate loop outside of the nested loops, thereby avoiding unwanted time-consuming operations.

Comparative results against the initial toolchain version, will be provided in section 4.2.



(a) Before refactor

(b) After refactor

Figure 3.12: Activity diagram showing the workflow of the function `custom_hook_windows_symbols` before (a) and after (b) refactoring

3.3.3 Memory utilization

Challenge encountered Regarding memory usage, the SCDG component of the toolchain poses a significant challenge. Large experiments that run for extended periods or utilize memory-intensive binaries result in the program consuming all available system memory, leading to device crashes. Consequently, conducting such experiments becomes impossible, restricting the toolchain's utility to relatively small experiments. This limitation severely hampers the current usability of the toolchain.

Investigation and solution To tackle this problem, first a study on the memory utilization of the program during execution had to be made. To conduct this analysis, the *memory-profiler* Python package was employed [39]. This tool offers a comprehensive overview of the program's memory usage over time, including insights into which functions and sections of code are responsible for the highest memory consumption.

Upon using this tool, it was discovered that the *claripy* [2] Python package, which is an abstracted constraint-solving wrapper used by *angr* [1], was not effectively clearing the cache containing the abstract syntax tree (AST) of the bit vectors representing the binary values being analysed, between two binary analyses. Consequently, the cache size continued to increase with each new bit vector creation, even if the bit vectors from previous binary analyses were no longer necessary. By forcibly resetting this cache through assignment to an empty dictionary, a substantial reduction in memory utilization over time was achieved. Additionally, employing the `del` instruction on the `SemaSCDG` object and invoking the Python garbage collector with `gc.collect()` at each iteration contributed to a slight decrease in the toolchain's memory consumption. Section 4.3 will present graphical illustrations depicting the effects of these optimizations on memory utilization.

3.4 Testing Methodology

In this Section, the testing methodologies that have been put into places in SEMA are presented. Subsection 3.4.1 introduces the unittest that have been integrated and used to verify the integrity of SEMA. Then, in Subsection 3.4.2, the Github CI/CD that have been added to the toolchain repository are presented.

3.4.1 Unittest

As a result of the refactor, the integration of a configuration file, the implementation of a new architecture, and various other enhancements previously outlined, it is imperative to verify whether the updated version of the toolchain remains consistent with the original version. To achieve this, a test suite has been crafted using Python's unittest module [18], specifically targeting the newly incorporated code within the SCDG and classifier components of the toolchain.

A. SCDG tests

The test suite comprises several components. Initially, it evaluates the REST API's interaction with SEMA's web application. This involves verifying the accurate transmission of SCDG arguments to the web application and ensuring that the application properly incorporates user inputs, sends them back to the SCDG, and updates the configuration file accordingly.

Additionally, the suite validates that program parameters specified in the configuration files are effectively utilized and that SCDG object attributes are appropriately adjusted based on the values from these files.

Consistency checks Furthermore, the test suite scrutinizes the consistency between the refactored toolchain and its original version. For this purpose, three directories have been established: one for the initial version, another for the refactored version, and a third for the version utilizing PyPy3. Each directory contains runs performed on the same dataset, which consists of syscall graphs for each analyzed binary. These runs are executed with identical parameters, including a 150-second timeout. To assess graph congruence, the suite leverages Python's networkx module [12] to determine if the graphs generated by the refactored version are isomorphic to those produced by the original version.

A similar comparison is conducted between the refactored version and the PyPy3-utilizing version. While checking the isomorphism between two graphs, if either version failed to finish the exploration within the specified timeout, the resulting graphs are deemed incomparable, as the outcome graph may vary slightly depending on the progress of exploration within the allotted time frame. Given the accuracy of these isomorphism tests, it can be concluded that both the refactored version of the toolchain and its PyPy3 variant represent enhancements over the original toolchain, while maintaining functionality.

Tutorial integration The original iteration of the toolchain included a Jupyter notebook offering a tutorial on utilizing the SCDG component. This tutorial has been seamlessly integrated into the refactored version, featuring updated commands. The continued compatibility of this tutorial with the new version serves as an effective means of validating its correctness against the initial version of the toolchain.

B. Classifier tests

In *SemaClassifier*, the sole components requiring testing are the REST API functionalities utilized for communication with SEMA's web application. The test suite focuses on evaluating the two functions comprising the API.

Initially, it verifies that *SemaClassifier* accurately transmits the available arguments to the web application. Subsequently, it ensures that the classifier correctly parses and incorporates the user-entered argument values received from the web application during program execution.

3.4.2 Github CI/CD

An enhancement to the initial toolchain involves the implementation of a GitHub workflow that conducts multiple checks and actions before each commit on the Git repository. Using the repository named *pre-commit-hooks* [40], a custom workflow has been built for the toolchain. This workflow performs various tasks, including trimming trailing whitespaces, ensuring files end with only a newline, attempting to load YAML, TOML, and JSON files to verify their syntax, prohibiting files larger than 500kB from being committed, detecting files containing merge conflict strings and preventing commits in such cases, verifying the proper formatting of JSON files and automatically fixing any discrepancies without altering the key order, and confirming that all Python files parse as valid Python code.

These comprehensive checks and actions serve to prevent errors when developers intend to commit updates to the toolchain's Git repository.

Chapter 4

Performance Analysis

After implementing the enhancements discussed in the preceding chapter, it becomes imperative to assess their impact on performance, focusing on execution time, memory utilization, and user reception. This chapter introduces the benchmarking techniques employed to analyze and compare the performance across different versions of the toolchain. Section 4.1 outlines the methodology used to conduct the benchmarking while Section 4.2 present results and discussions concerning execution time and exploration performance. Memory utilization implications are addressed in Section 4.3, while user feedback on the new version is presented in Section 4.4.

4.1 Benchmark methodology

Setup To conduct the performance analysis of the SCDG component within the toolchain, a dataset comprising seven distinct malware families, *FeakerStealer* [8], *RedLineStealer* [13], *autoit* [9], *ircbot* [22], *shiz* [26], *stormattack* [10] and *upatre* [23], each containing approximately 30 binaries, was utilized.

Ensuring methodological consistency, we employed the same dataset and toolchain versions across all analyses. The experiments were conducted with specific parameters: a timeout of 150 seconds, employing the CDFS exploration technique, activating the `count_block` feature to gather information on the number of instructions and blocks visited, and enabling the use of plugins. The remaining parameters were set to their default values as specified in *configs/default_config.ini.*

The data presented in this chapter are derived from executions conducted on a virtual machine operating Ubuntu 20.04. The virtual machine is equipped with 4 CPUs, 2x16 GiB of memory, and utilizes an x86_64 architecture.

Data collection To collect time performance metrics, the Python *time* module [17] is employed at specific locations within the codebase to measure execution time, exploration time, and hooking time. Whenever a syscall is encountered during the exploration of a binary, it is logged into a dictionary. This accumulation enables the calculation of

the total number of syscalls found and the count of unique syscalls identified at the conclusion of the process.

To analyze memory utilization during a run of the *SemaSCDG*, the Python package *memory-profiler* [39] is employed. This package offers a straightforward approach to capturing memory snapshots during program execution, facilitating the examination of the program's memory usage at runtime.

Organization An examination is conducted on three iterations of the toolchain: the original version, the Python 3-refactored edition, and the PyPy3-refactored variant, scrutinizing various dimensions. Section 4.2 delves into performance analysis, while Section 4.3 concentrates on memory utilization.

4.2 Performance comparison

This section maintains a consistent structure across all figures, comparing three iterations of the toolchain. The initial version is depicted in blue, followed by the refactored version utilizing Python 3 in orange, and finally, the refactored version employing PyPy3, represented in green.

Execution time and hooking time Initially, we'll examine the impact of the performance enhancements detailed in Chapter 3.3.1 on the average execution time for each binary family. The execution time for a single binary refers to the elapsed time starting from the initialization of the angr project to the completion of the exploration process. The following graphs illustrate the resulting changes.

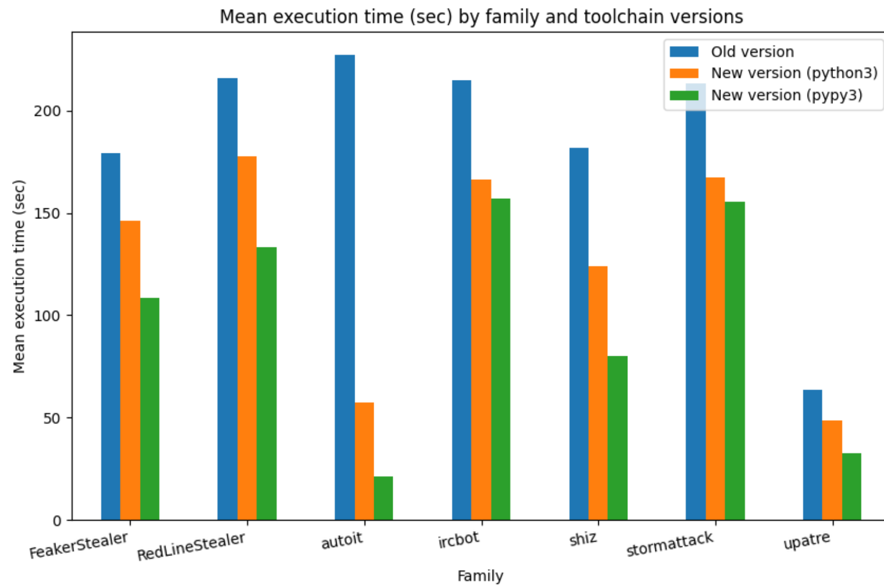


Figure 4.1: Graph of the mean execution time in seconds by family and toolchain version

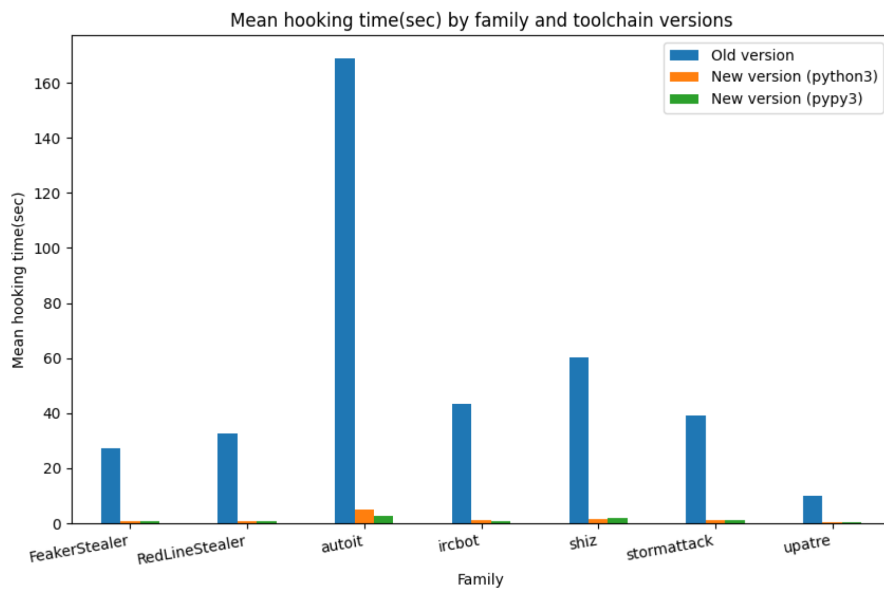


Figure 4.2: Graph of the mean hooking time in seconds by family and toolchain version

The Figure 4.1 illustrates that the refactoring of the initial version, depicted in orange, resulted in significant enhancements in execution time. This improvement stemmed from both minor optimizations introduced during the refactoring process and notably reduced hooking time by more than 95% for each family, as demonstrated in Figure 4.2.

Furthermore, employing PyPy3 in the new version further augmented performance by incorporating the JIT compiler. The total improvement in terms of execution time goes from 26.6% for the *ircbot* family to 90.7% for the *autoit* family. Such enhancements in execution time will enable future users to conduct experiments more swiftly, thereby enhancing productivity.

Syscalls found Moving forward, we will examine the average count of distinct syscalls discovered per family across the various versions of the toolchain.

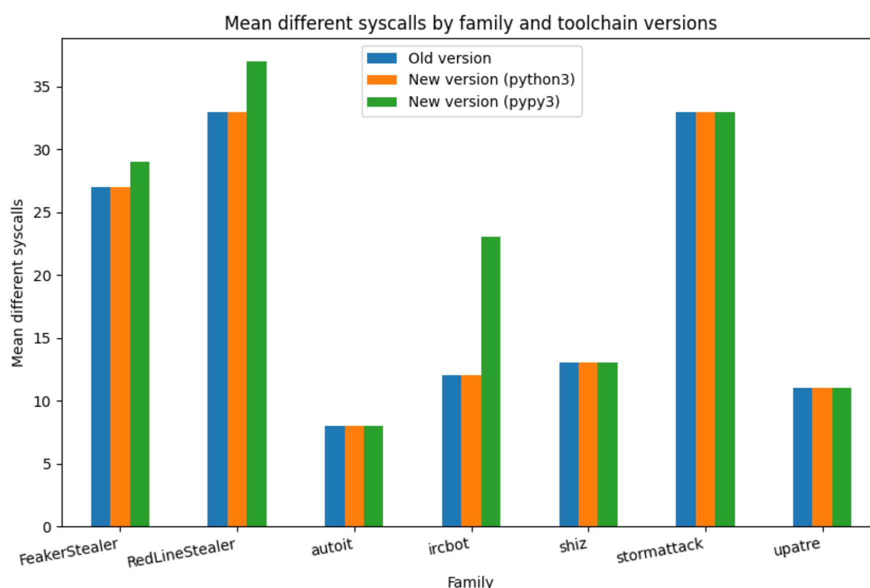


Figure 4.3: Graph of the mean different syscalls found by family and toolchain version

Figure 4.3 demonstrates that refactoring the version using Python 3 did not impact the number of distinct syscalls identified. Given that the primary improvement from the refactoring was in hooking time, these findings are not surprising and indicate that the refactor preserved the original toolchain's functionality.

In contrast, employing PyPy3 resulted in the discovery of more distinct syscalls within the *FeakerStealer*, *RedLineStealer*, and *ircbot* families. This can be attributed to PyPy3 enabling faster binary exploration within the same time constraints. It effectively showcases the positive impact of integrating PyPy3 with the SCDG component of the SEMA toolchain.

Blocks visited Another intriguing aspect to consider is the number of address blocks traversed during the exploration of the binaries. Figure 4.4 below depicts the average count of distinct blocks visited by malware families. It's noticeable that for the three families where the version utilizing PyPy3 discovered more syscalls, they also traversed more blocks, corroborating the notion that the exploration could delve deeper into the binary. Similarly, the same trend is observed in other families, although in these cases, no new syscalls were identified in the newly explored segments of the binaries.

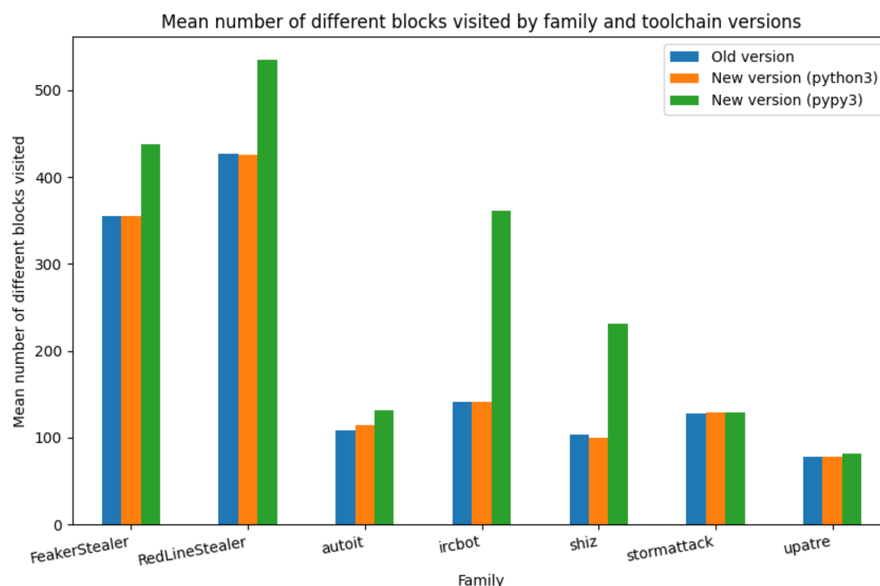


Figure 4.4: Graph of the mean different blocks visited by family and toolchain version

Completely explored binaries The final aspect we can examine is the percentage of binaries explored in their entirety, categorized by family and toolchain version. In this context, a binary is considered fully explored when the exploration method, such as CDFS, halts before reaching the timeout. The Figure 4.5 illustrates this metric.

Upon analyzing the figure, we observe that for families like *FeakerStealer* and *upatre*, *SemaSCDG* can now completely explore a higher proportion of binaries compared to previous versions. Notably, the *RedLineStealer* family's percentage of completely explored binaries jumps from 0% to approximately 55%. This underscores the significant impact of integrating PyPy3 on the exploration performance of the toolchain.

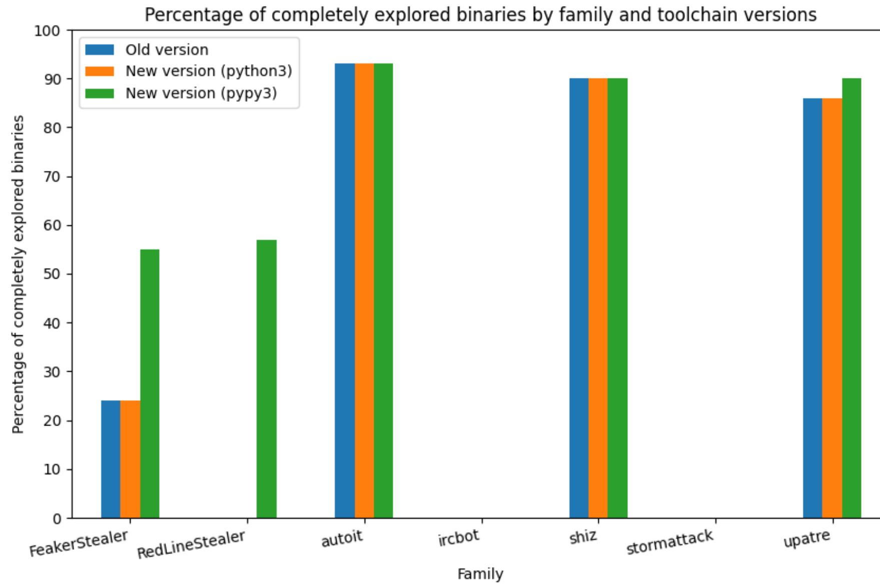


Figure 4.5: Graph of the percentage of completely explored binaries by family and toolchain version

Logs impact Considering that the logging system of *SemaSCDG* underwent modifications during the refactor, we aim to assess the effects of activating and deactivating logs during a single run on our benchmark dataset. Figure 4.6 depicts the mean execution time per family for the refactored version employing PyPy3 under these two conditions.

Notably, the graph reveals only marginal impacts on execution time due to the presence of logs. Given the inherent variability in execution times across runs, it's reasonable to conclude that enabling or disabling logs does not significantly influence experiment execution times.

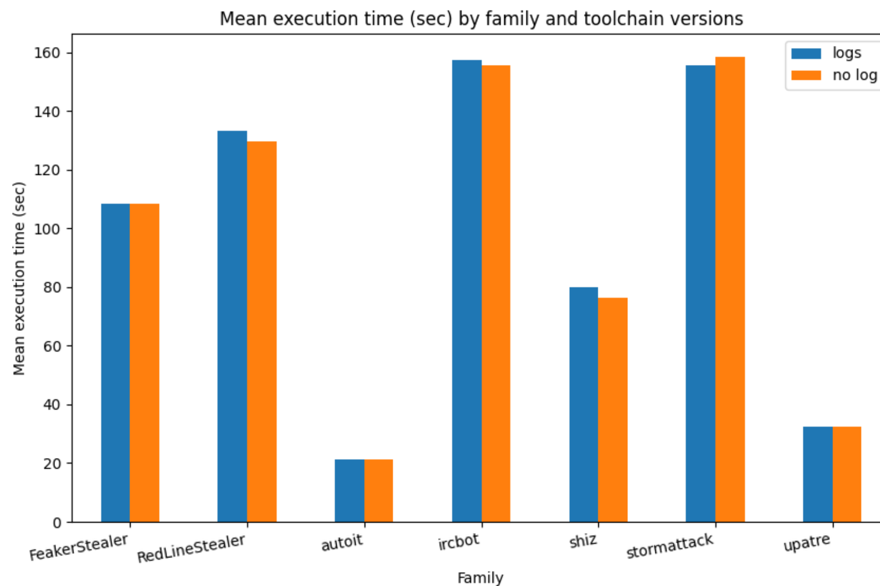


Figure 4.6: Graph of the mean execution time by family with the logs activated and deactivated on the new toolchain version using PyPy3

Plugins impacts The refactored version of the toolchain introduces the capability for users to selectively enable or disable various plugins for binary exploration, whereas previously, these plugins were consistently activated. Upon investigating the influence of these plugins, it became apparent that the graphs resulting from a run with the plugins deactivated failed to pass the same type of isomorphism test conducted in Section 3.4 when juxtaposed with the outcomes of the original toolchain version.

4.3 Memory Utilization

4.3.1 Before memory improvements

The memory utilization of the three toolchain versions was analyzed prior to implementing the memory improvements outlined in section 3.3.3. Employing the `mprof run` command for each version produced a *.dat* file containing memory snapshots taken at constant intervals during execution. Subsequently, a graph was generated using the `mprof plot` command on the three generated *.dat* files. The resulting graph is displayed below.

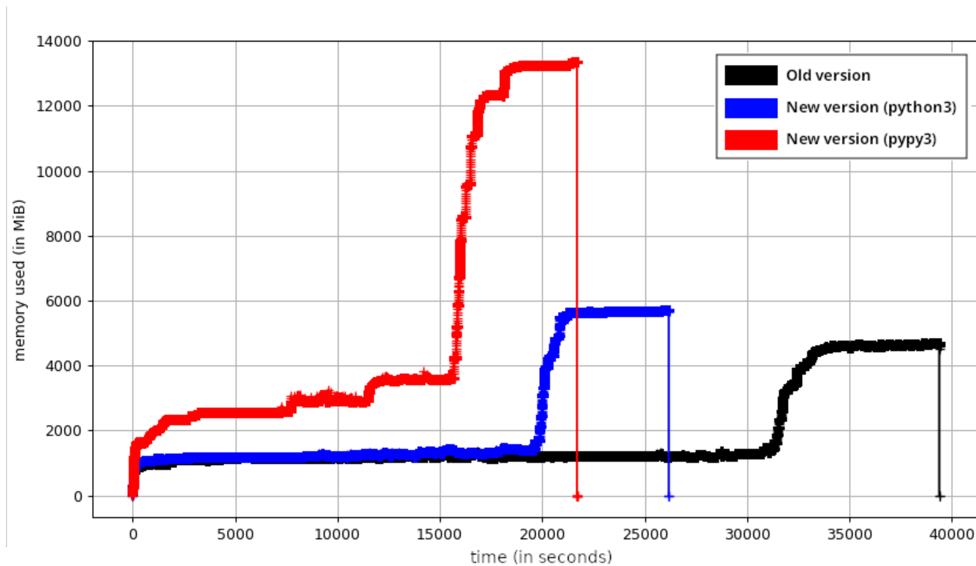


Figure 4.7: Memory utilization in MiB over time (in seconds) for the three different version of the toolchain before memory improvements were made

Figure 4.7, representing the memory utilisation over time for the three different versions, illustrates several key observations.

Firstly, notable improvements in total execution time are apparent. The initial version of the toolchain, in black, requires approximately 40,000 seconds, equivalent to 11 hours, to complete the experiment.

In contrast, the refactored version, in blue, accomplishes it in just over 25,000 seconds, roughly less than 7 hours, while the PyPy3 version, in red, takes around 22,000 seconds, approximately 6 hours which represents a 45% improvement on the total experiment time. However, the most crucial aspect here is the memory consumption across different versions. While the initial toolchain version consumes no more than 4500 MiB, the PyPy3 version peaks at 135,000 MiB, which represents a 2900% increase, highlighting the memory challenges discussed in 3.3.3, exacerbated by the increased memory requirements of PyPy3 when generating JIT code.

Additionally, the graph depicts a continual increase in memory usage throughout the experiment, indicating a persistent upward trend.

4.3.2 After memory improvements

Following the implementation of the memory enhancements suggested in section 3.3.3, identical measurements were conducted on the two revised versions of the toolchain. The updated outcomes are depicted in the plot below.

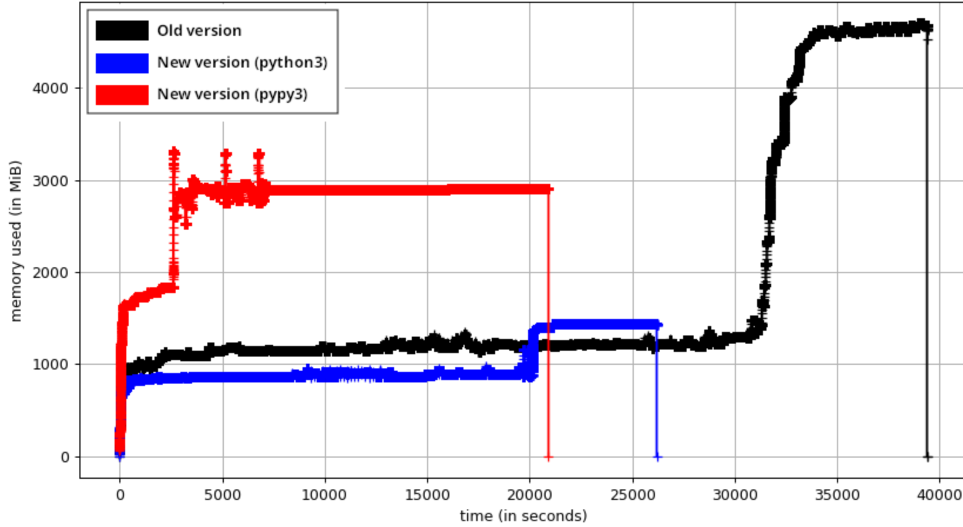


Figure 4.8: Memory utilization in MiB over time (in seconds) for the three different version of the toolchain after memory improvements were made

Figure 4.8 clearly demonstrates the significant impact of the implemented improvements on memory utilization during runtime. Presently, the refactored version using Python3, in blue, consistently remains below 1500 MiB, surpassing the initial version of the toolchain, in black, which still hovers just below 4500 MiB, with a 66.6% improvement.

Similarly, the version utilizing PyPy3, in red, maintains a level around 3000 MiB, a considerable improvement compared to its performance without the enhancement. Notably, variations in memory usage correspond to the freeing of memory during execution.

Although the PyPy3 version continues to consume more memory than the Python3 version due to the generation of JIT code, it still outperforms the initial toolchain version by improving the memory utilisation by 33.3%. Moreover, the memory requirements of the JIT code stabilize over time, as they are constrained by the amount of code utilized during execution.

4.4 Users Evaluation

To conduct a user evaluation, Christophe Crochet, Serena Lucca, Samy Bettaieb, and Charles-Henry Bertrand Van Ouytsel, all contributors to the SEMA toolchain, were asked to test the new version of SEMA, with particular emphasis on the updates introduced in the SCDG component. The key insights from their reviews are summarized below. The entire users evaluation is available in Appendix B.

4.4.1 Installation Process

Pros	Cons
Much easier	Pypi only install requirements
Readme self-explanatory	Long
Centralized in a simpler/clearer Makefile	Non-parameterized
Dockerhub image available	
Pypi package to install requirements	

Table 4.1: Pros and cons of the installation process of the new version of SEMA

Overall, Table 4.1 shows that the installation process is now easier due to better documentation and centralization. The images being available on Dockerhub and the Pypi package are a plus. For now, the Pypi package only installs the dependencies of the project and could be enhance in the future. The installation process is still long due to the complexity of the project. Another improvement that could be made is to be able to choose exactly which components of *SemaSCDG* are included in the installation.

4.4.2 Utilization

Pros	Cons
Good speed-up	
Clear logs	But could be more manageable
Structure clear	
Good management of file input/ folder	Input path could be better managed
Tutorial clear	
Recording of parameters	
Config file allow easy fine tuning of experiments	

Table 4.2: Pros and cons of the Utilization of the new version of SEMA

Table 4.2 shows that SemaSCDG is now easier to use overall. Two points that could be improved, are the logs that could be more manageable by adding options to modify the level of angr and SEMA logs independently, and the processing of input path that could be better managed.

4.4.3 Adding functionalities

Pros	Cons
The refactor makes it easier	Whole application not mounted
Microservices allow to easily add services	
Quite straightforward	Could be easier with a README
Good refactor of Exploration	
to add techniques	

Table 4.3: Pros and cons of adding functionalities of the new version of SEMA

Table 4.3 shows that the users find it easier to add functionalities in the new version of the toolchain but could be further enhance by the addition of a README explaining the process. Moreover, the whole application is not mounted automatically in the container which disables the possibility to modify the code when the container is running. Documentation about how to mount the entire application has been added.

4.4.4 Overall Comparison

Advantages over Old Version	Advantages of Old Version
More efficient to extend	Passing individual input file in command line
Cleaner code	
Easier to install	
Faster	
Maintainable	
More usable	
More documented	

Table 4.4: Overall comparison between the old and new version of SEMA

Table 4.4 shows that, overall, the new version of SEMA is more efficient in terms of performance but also more maintainable and easy to use. However, one advantage that has been lost is the ability to quickly run small experiments using the command line without needing configuration files. It would be beneficial to reintroduce this capability in the future.

Suggestions for Improvement Below are the users' suggestions to improve the toolchain, providing valuable guidance for potential future work. Notable recommendations include adding multithreading, developing a Windows version of SEMA, and creating a web version for public use, as detailed in Chapter 5. The issue of reducing

logs from angr has already been addressed in a subsequent iteration following the user evaluation.

- Developing Unit tests for Simprocedure
- Developing brief tutorials to expand SEMA
- Better Pypi with full toolchain and not only requirements
- More configurable installation
- More CI/CD
- Multithreading
- Windows version of SEMA
- Web version for grand public/Secured
- Reduce logs from angr

Chapter 5

Future Work

In this chapter, potential improvements for the SEMA-toolchain are discussed, offering suggestions and insights on areas that could benefit from enhancement.

Classifier enhancements To further advance the micro-services architecture and enhance modularity, the structure of *SemaClassifier* could be improved. Currently, the different machine learning models are implemented as separate classes in different folders within the codebase. An improvement could involve creating new micro-services, each containing a single machine learning model.

This would provide better separation of code logic, allow users to select the models necessary for their use of the toolchain, simplify the addition of new models, and facilitate the maintenance of existing models. However, excessive splitting into micro-services could lead to unnecessarily complex code, so this approach should be considered carefully.

Toolchain dependencies The SEMA toolchain has been tested and implemented for Python 3.8. One potential improvement is updating this dependency to a later version of Python, which may include enhancements related to the Global Interpreter Lock (GIL).

Additionally, the toolchain is currently designed to run on Ubuntu 20.04. Expanding support to include other versions of Ubuntu or even different operating systems, such as Windows, could broaden the user base. Furthermore, the toolchain does not utilize the most recent versions of its dependent libraries. While updating these libraries could introduce beneficial improvements, it must be approached with caution to avoid potential bugs and performance regressions.

Use multiple exploration techniques Another avenue worth exploring is the ability to employ multiple exploration techniques simultaneously during binary exploration. The angr framework allows stacking exploration techniques by employing multiple calls to `use_technique(exploration_technique)` on the simulation manager. When two calls to `use_technique(exploration_technique)` are initiated, angr executes the

first exploration technique followed by the second, alternating between them throughout the exploration process.

Crucially, for this approach to function correctly, the exploration techniques must not interfere with each other. If both techniques utilize the `step` function, there's a risk of states being missed based on each technique's behavior, necessitating cautious application. Similarly, if both techniques employ `filter` functions, it's essential to recognize that the filter function of the first technique precedes that of the second. Presently, within the toolchain, stacking exploration techniques without interference is unattainable, as they weren't initially designed for this purpose. Consequently, a redesign of exploration techniques is imperative to facilitate this capability.

Multi-threading To enhance the performance of the SCDG, we explored transitioning from the current single-threaded implementation to a multi-threaded one. However, upon implementing a test version, we did not observe significant performance improvements, primarily due to the constraints imposed by the Global Interpreter Lock (GIL), as discussed in Section 3.3.1.

One potential solution could involve migrating from the current Python version 3.8 to the latest version, which offers options to avoid the GIL and potentially yield further performance enhancements. Integrating multi-threading fully into the SCDG component of the toolchain would necessitate a substantial overhaul of the approach to angr binary exploration to properly implement thread coordination.

Regrettably, due to time constraints, this overhaul was not carried out within the scope of this master's thesis. However, it presents an avenue for future development work.

Database support Another upgrade to consider for future work is the addition of database support. Currently, all inputs taken by the toolchain are stored in the user's local files. Since these inputs can include malware, storing them locally poses a risk if the binaries are accidentally executed. A database could securely store these binaries, preventing such mistakes by allowing the toolchain to retrieve binaries from the database.

Additionally, a database could centralize the available binaries, machine learning models, and SCDGs, making it easier to share experimental results. However, this integration should be carefully considered, as it could introduce malicious behavior and alter the toolchain's purpose.

Use of Numba Integrating Numba into the toolchain could result in significant performance enhancements. In the context of *SemaSCDG*, Numba was unsuitable due to its lack of support for custom types, such as those used in angr.

However, Numba could be effectively utilized in the Classifier component of the toolchain, which relies on numpy and performs extensive numerical computations. Incorporating Numba into the toolchain is straightforward, as its decorator system allows for easy implementation. By adding these decorators to functions that perform heavy

numerical computations, the process can be accelerated through Just-In-Time (JIT) compilation. [27]

Better web interface In conjunction with adding database support, the web application of SEMA could be refined. Currently, the interface is basic and provides limited feedback on background processes. The web interface could offer enhanced functionalities, such as visualizing run results stored in the database through graphs or statistics. Additionally, the interface design could be modernized to improve user experience and make navigation more intuitive and enjoyable.

Chapter 6

Conclusion

In conclusion, this master thesis documents the transformation of the SEMA toolchain architecture from a monolithic to a micro-services-based design, greatly simplifying the addition of new functionalities for developers. Significant enhancements have been made to improve maintainability, including streamlined installation processes, comprehensive documentation, and customizable configuration files for *SemaSCDG* parameters.

Refactoring has enhanced the codebase quality, reducing hooking time by 95% for each tested family. Performance improvements have been achieved through the integration of PyPy3 into *SemaSCDG*, resulting in execution time enhancements ranging from 26% to 90%, depending on the family. Memory usage has also been optimized by addressing leaks, with PyPy3 yielding a 33% improvement and Python3 a 66% improvement over the initial version.

Testing has confirmed that these modifications do not compromise the toolchain's functionality. User evaluations have validated the enhancements, highlighting the increased utility and positive impact of the work on SEMA.

These advancements render the SEMA toolchain more robust and user-friendly, enhancing its maintainability and usability. Consequently, SEMA is now better equipped to support future research in malware analysis and attract a wider user base across various contexts. As malware threats escalate, SEMA offers an innovative approach by combining symbolic execution analysis with machine learning to combat emerging threats. When integrated with existing static and dynamic analysis tools like Yara and Cuckoo, SEMA enables more detailed malware analysis and the creation of more precise signatures to counteract these threats.

Bibliography

[1] The angr Project contributors. angr. https://github.com/angr/angr. Accessed: 26-02-2024.

[2] The angr Project contributors. claripy. https://github.com/angr/claripy?tab=readme-ov-file. Accessed: 26-02-2024.

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), may 2018.

[4] C.H. Bertrand Van Ouytsel, C. Crochet, K.H.T. Dam, and A. Legay. Sema toolchain. https://github.com/csvl/SEMA-ToolChain.

[5] Michael Brengel and Christian Rossow. YARIX: Scalable YARA-based malware intelligence. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3541–3558. USENIX Association, August 2021.

[6] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56:82–90, 02 2013.

[7] Eric Cheng. *Binary analysis and symbolic execution with angr*. PhD thesis, PhD thesis, The MITRE Corporation, 2016.

[8] Ben Cohen. Fickerstealer. https://www.cyberark.com/resources/threat-research-blog/fickerstealer-a-new-rust-player-in-the-market, 2021.

[9] Autoit consulting ltd. Autoit. https://www.autoitscript.com/site/, 2024.

[10] HYPR Corp. Storm worm. https://www.hypr.com/security-encyclopedia/storm-worm, 2024.

[11] Arthur Crapé and Lieven Eeckhout. A rigorous benchmarking and performance analysis methodology for python workloads. In *IEEE International Symposium on Workload Characterization, IISWC 2020, Beijing, China, October 27-30, 2020*, pages 83–93. IEEE, 2020.

[12] NetworkX developers. networkx. https://networkx.org/, 2024.

[13] The Florida Center for Cybersecurity. Redlinestealer. https://cyberflorida.org/redline-stealer-malware-analysis/, 2023.

[14] Django Software Foundation and individual contributors. Django. https://github.com/django/django/tree/main, 2024.

[15] Python Software Foundation. Can't we get rid of the global interpreter lock? https://docs.python.org/3/faq/library.html#can-t-we-get-rid-of-the-global-interpreter-lock, 2024.

[16] Python Software Foundation. configparser. https://docs.python.org/3/library/configparser.html, 2024.

[17] Python Software Foundation. time. https://docs.python.org/3/library/time.html, 2024.

[18] Python Software Foundation. unittest. https://docs.python.org/3/library/unittest.html, 2024.

[19] Martin Fowler. Refactoring: Improving the design of existing code. In Don Wells and Laurie A. Williams, editors, *Extreme Programming and Agile Methods - XP/Agile Universe 2002, Second XP Universe and First Agile Universe Conference Chicago, IL, USA, August 4-7, 2002, Proceedings*, volume 2418 of *Lecture Notes in Computer Science*, page 256. Springer, 2002.

[20] Docker Inc. Docker compose quickstart. https://docs.docker.com/compose/gettingstarted/. Accessed: 10-11-2023.

[21] Docker Inc. Docker. https://www.docker.com/, 2024.

[22] Trend Micro Incorporated. Ircbot. https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/ircbot, 2024.

[23] Trend Micro Incorporated. Upatre. https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/upatre, 2024.

[24] The institute of electrical and electronics engineer in New York. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[25] Sainadh Jamalpur, Yamini Navya, Perla Raja, Gampala Tagore, and G. Rao. Dynamic malware analysis using cuckoo sandbox. pages 1056–1060, 04 2018.

[26] AO Kaspersky Lab. Shiz. https://threats.kaspersky.com/fr/threat/Backdoor.Win32.Shiz/, 2024.

[27] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery.

[28] Lorenzo De Lauretis. From monolithic architecture to microservices architecture. In Katinka Wolter, Ina Schieferdecker, Barbara Gallina, Michel Cukier, Roberto Natella, Naghmeh Ramezani Ivaki, and Nuno Laranjeiro, editors, *IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops 2019, Berlin, Germany, October 27-30, 2019*, pages 93–96. IEEE, 2019.

[29] Yu Liu, Xu Zhou, and Wei-Wei Gong. A survey of search strategies in the dynamic symbolic execution. In *ITM Web of Conferences*, volume 12, page 03025. EDP Sciences, 2017.

[30] Serena Lucca, Christophe Crochet, Charles-Henry Bertrand Van Ouytsel, and Axel Legay. On exploiting symbolic execution to improve the analysis of rat samples with angr. In *Foundations and Practice of Security: 16th International Symposium, FPS 2023, Bordeaux, France, December 11–13, 2023, Revised Selected Papers, Part I*, page 339–354, Berlin, Heidelberg, 2024. Springer-Verlag.

[31] Mark Lutz. *Programming Python - Powerful Object-Oriented Programming: Covers Python 3.x (4. ed.)*. O'Reilly, 2011.

[32] A. Radovan M. Šipek, B. Mihaljević. Exploring aspects of polyglot high-performance virtual machine graalvm. `https://doi.org/10.23919/MIPRO.2019.8756917`, 2019.

[33] Manon Oreins. Fork of the sema toolchain linked to this master thesis. `https://github.com/Manon-Oreins/SEMA-ToolChain/tree/refactor_simproc`, 2023.

[34] Manon Oreins. Dockerhub semascdg image. `https://hub.docker.com/r/manonoreins/sema-scdg`, 2024.

[35] Manon Oreins. Pypi sema package. `https://pypi.org/project/sema-toolchain/`, 2024.

[36] R Oudkerk. cpython multiprocessing. `https://github.com/python/cpython/tree/3.12/Lib/multiprocessing/`, 2008.

[37] Charles-Henry Bertrand Van Ouytsel, Christophe Crochet, Khanh-Huu-The Dam, and Axel Legay. Tool paper - SEMA: symbolic execution toolchain for malware analysis. In Slim Kallel, Mohamed Jmaiel, Mohammad Zulkernine, Ahmed Hadj Kacem, Frédéric Cuppens, and Nora Cuppens, editors, *Risks and Security of Internet and Systems - 17th International Conference, CRiSIS 2022, Sousse, Tunisia, December 7-9, 2022, Revised Selected Papers*, volume 13857 of *Lecture Notes in Computer Science*, pages 62–68. Springer, 2022.

[38] Charles-Henry Bertrand Van Ouytsel and Axel Legay. Analysis and classification of malware based on symbolic execution and machine learning. page 168, 2024.

[39] Fabian Pedregosa. memory-profiler. `https://github.com/pythonprofilers/memory_profiler`, 2014. Accessed: 15-03-2024.

[40] Ken Struys pre-commit dev team: Anthony Sottile. Github cicd hooks. `https://github.com/pre-commit/pre-commit-hooks`, 2014.

[41] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996.

[42] Alexander Roghult. Benchmarking python interpreters : Measuring performance of cpython, cython, jython and pypy. 2016.

[43] Jean Bosco. Rwibutso. Microservices approach to build scalable and distributed systems, louvain school of management, université catholique de louvain. `http://hdl.handle.net/2078.1/thesis:25188`, 2022.

[44] The PyPy Team. Pypy. `https://www.pypy.org/`, 2024.

[45] Fish Wang and Yan Shoshitaishvili. Angr - the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9, 2017.

[46] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *Proceedings - 2002 IEEE International Conference on Data Mining, ICDM 2002*, Proceedings - IEEE International Conference on Data Mining, ICDM, pages 721–724, 2002. 2nd IEEE International Conference on Data Mining, ICDM '02 ; Conference date: 09-12-2002 Through 12-12-2002.

[47] Qi Zhang, Ling Liu, Calton Pu, Qiwei Dou, Liren Wu, and Wei Zhou. A comparative study of containers and virtual machines in big data environment. pages 178–185, 07 2018.

Appendix A

Additional graphs

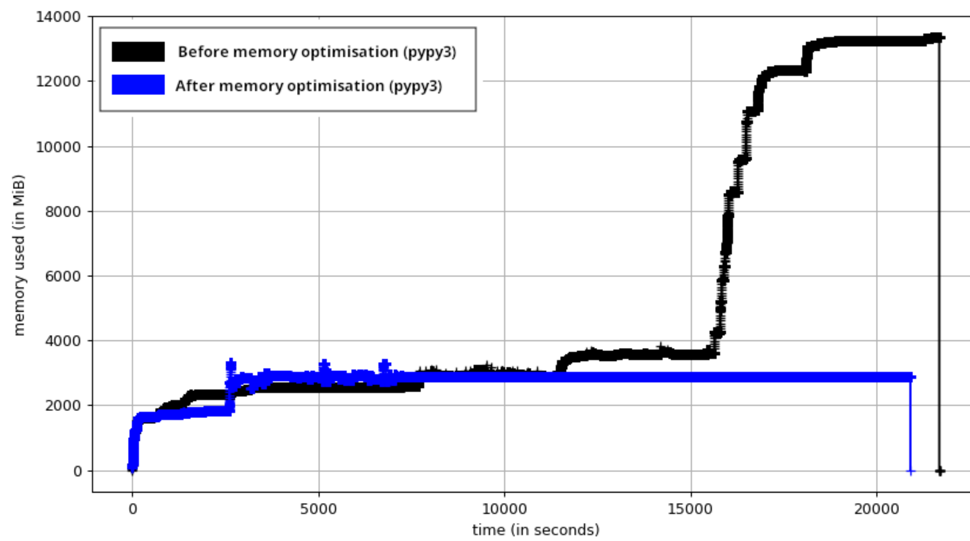


Figure A.1: Graph of the memory utilisation of two version of the toolchain, before and after the memory improvement done in 3.3.3, using pypy3

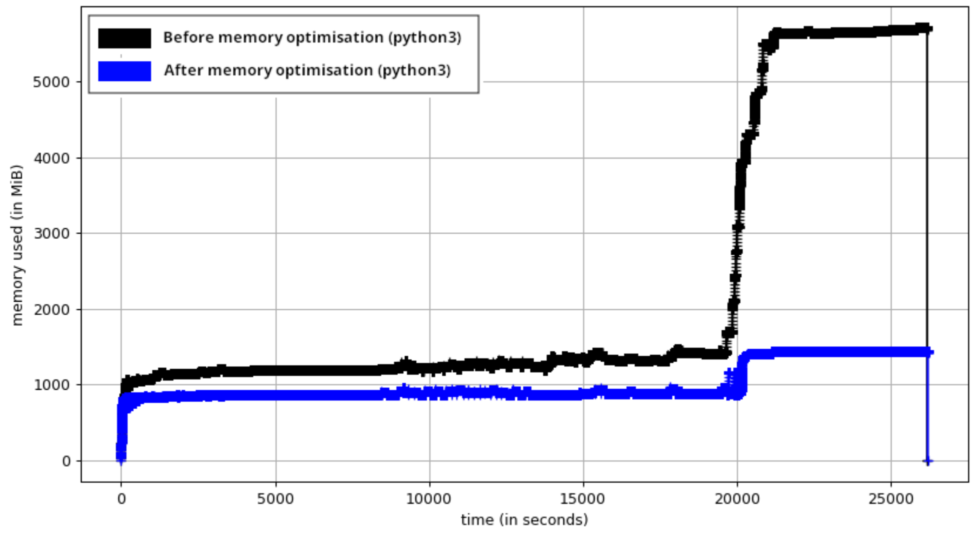


Figure A.2: Graph of the memory utilisation of two version of the toolchain, before and after the memory improvement done in 3.3.3, using python3

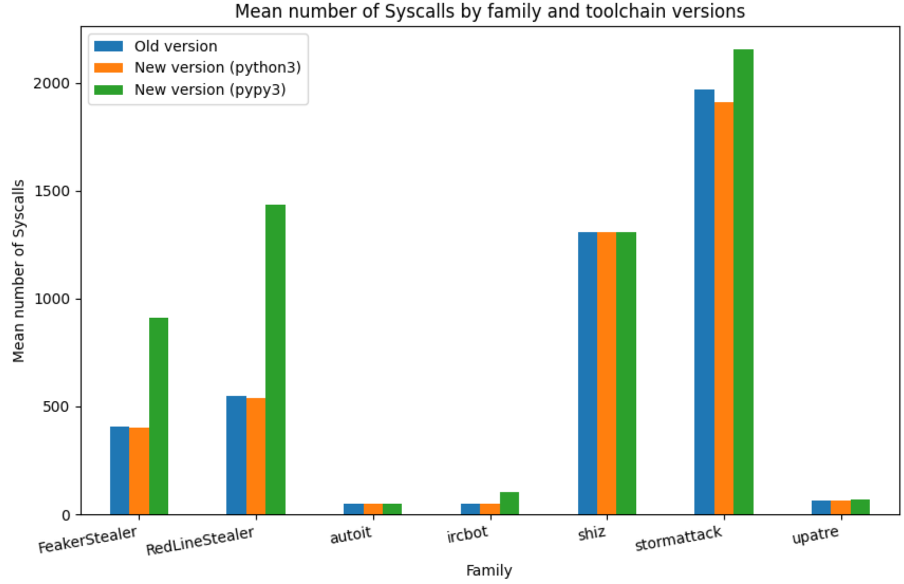


Figure A.3: Graph of the mean number of syscalls found by family and toolchain version

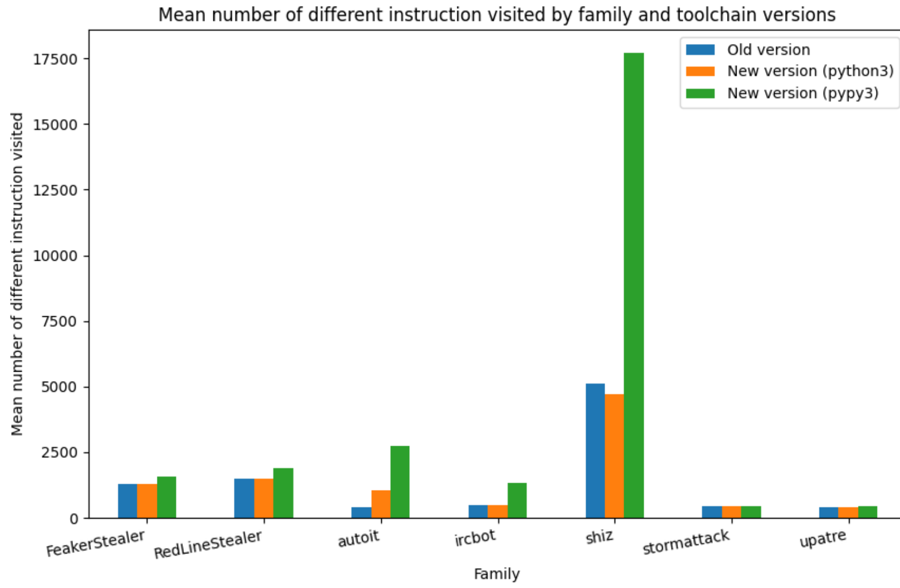


Figure A.4: Graph of the mean different instructions visited by family and toolchain version

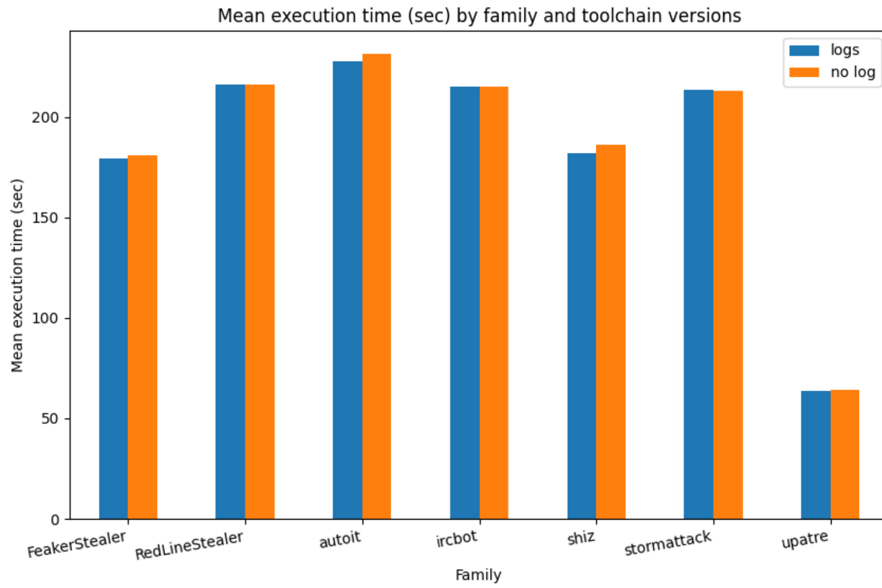


Figure A.5: Graph of the mean execution time by family with the logs activated and deactivated on the old toolchain version

Appendix B

Users evaluation

B.1 From Serena Lucca

B.1.1 Installation Process

Pros	Cons
Good readme	Fix the command to start the docker: manonoreins-sema-scdg into manonoreins/sema-scdg

Table B.1: Pros and cons of the installation process of the new version of SEMA

B.1.2 Utilization

Pros	Cons
Utilization is easier with the config file no need to remember long command lines	Finding the right path to put in the config file to analyze one file or a folder containing multiple files is a bit complicated with all the folders and subfolders needed

Table B.2: Pros and cons of the Utilization of the new version of SEMA

B.1.3 Adding functionalities

Pros	Cons
The refactor makes it easier to modify the code	Mounting the whole application file would be nice to modify the code while the container runs

Table B.3: Pros and cons of adding functionalities of the new version of SEMA

B.1.4 Overall Comparison

Advantages over Old Version	Advantages of Old Version
Cleaner code, easier to use and to install	

Table B.4: Overall comparison of the two versions of SEMA

Suggestions for Improvement Reduce logs from agnr, especially ticked state

Additional Comments Great job! Sema is much more user friendly and looks more serious now.

B.2 From Samy Bettaieb

B.2.1 Installation Process

Pros	Cons
Clear readme	Fix the command to start the docker: manonoreins-sema-scdg into manonoreins/sema-scdg

Table B.5: Pros and cons of the installation process of the new version of SEMA

B.2.2 Utilization

Pros	Cons
Config file makes setting all the parameters easy	Notebook didn't work for me (might a problem on my end, or there is a lack of info on the readme?)

Table B.6: Pros and cons of the Utilization of the new version of SEMA

B.2.3 Adding functionalities

Pros	Cons
Seems easy to add new exploration strategy	Maybe a short readme on how to exploration strategies? (explain that you have to implement a new class, with the step method, ...)

Table B.7: Pros and cons of adding functionalities of the new version of SEMA

B.2.4 Overall Comparison

Advantages over Old Version	Advantages of Old Version
Nice refactoring and new project structure Pypi, faster	Command line without config file was good too for individual files if you don't have to setup a lot of parameters.

Table B.8: Overall comparison of the two versions of SEMA

Suggestions for Improvement A readme or tuorial on how to add things like exploration strategies or simprocedures

Additional Comments Good job, SEMA looks great. Like video games, SEMA is good for mental health

B.3 From Charles-Henry Bertrand Van Ouytsel

B.3.1 Installation Process

Pros	Cons
Efficient docker file readme self-explanatory	No specific issue

Table B.9: Pros and cons of the installation process of the new version of SEMA

B.3.2 Utilization

Pros	Cons
Config file is great and allow easy fine tuning of experiments recording of parameters Good speed-up during running Clear log Structure clear Good management of file input/ folder Tutorial clear	Full path could be a bit annoying to put un congig file for individual samples Maybe distinguishing general INFO and INFO related to API call could be great (to reduce verbosity)

Table B.10: Pros and cons of the Utilization of the new version of SEMA

B.3.3 Adding functionalities

Pros	Cons
Quite straightforward to add simprocedure or plugins Good refactor of Exploration to add techniques	Maybe add brief readme/wiki on how to add simproc/plugins/Explor method?

Table B.11: Pros and cons of adding functionalities of the new version of SEMA

B.3.4 Overall Comparison

Advantages over Old Version	Advantages of Old Version
Good refactoring making the code more efficient to extend (e.g.: exploration) but also in its usage (e.g.: hooking process before symbolic execution). Installation is now straightforward	Passing individual input file in command line

Table B.12: Overall comparison of the two versions of SEMA

Suggestions for Improvement Developping Unit tests for Simprocedure, implementing CI/CD on repo and developping brief tutorials to expand SEMA

Additional Comments Good work! It makes SEMA more operational for new users !

B.4 From Christophe Crochet

B.4.1 Installation Process

Pros	Cons
Much more easier all centralized in a simpler/clearer makefile Dockerhub image available Pypi package available for installing requirements of the project	long (but could be expected due to the size of the project) non-parametrizable (i.e what if i don't want cuda, pypy3, etc.) pypi only install requirements

Table B.13: Pros and cons of the installation process of the new version of SEMA

B.4.2 Utilization

Pros	Cons
The file structure of the project is greatly improved! Thus allowing to dev new feature more easily Configuration files and memory leak fixes allow better usage for large scale experiments Same for the parameterization of the logs The improved performance is also very convenient Dockerhub is very nice to share our tool	Except some difficulties already present before but out of scope of this thesis, none (path annoying to select)

Table B.14: Pros and cons of the Utilization of the new version of SEMA

B.4.3 Adding functionalities

Pros	Cons
Micro services architecture allows to easily add new services without modifying too much, we will try soon with old student thesis Config file and large scale experiment scripts are a plus!	parameterization of the installation could allow to not install unneeded module

Table B.15: Pros and cons of adding functionalities of the new version of SEMA

B.4.4 Overall Comparison

Advantages over Old Version	Advantages of Old Version
Maintainable More usable More documented	

Table B.16: Overall comparison of the two versions of SEMA

Suggestions for Improvement

- Better pipy with full toolchain and not only requirements
- More configurable installation (no pypy/cuda, etc. for example)
- Windows version of SEMA (but out of scope of the thesis)
- Web version for grand public/Secured (but out of scope of the thesis)
- Multithreading (but out of scope of the thesis)
- More CI/CD
- More unit test (but out of scope of the thesis)

Additional Comments Great works !

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**