Forensic Science International: Digital Investigation 53 (2025) 301920

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS USA 2025 - Selected Papers from the 25th Annual Digital Forensics Research Conference USA

Memory Analysis of the Python Runtime Environment

Hala Ali [a,*], Andrew Case [b], Irfan Ahmed [a]

[a] Department of Computer Science, Virginia Commonwealth University, USA
[b] Volatility Foundation, USA

ARTICLE INFO

Keywords:
Memory forensics
Python runtime
Malware
Volatility 3

ABSTRACT

Memory forensics has become a crucial component of digital investigations, particularly for detecting sophisticated malware that operates solely in system memory without leaving traces on the file system. Although memory forensics provides a complete view of the system state during acquisition, prior research efforts have primarily focused on analyzing kernel-level data structures for malware detection. With the propagation of kernel-level malware, operating system vendors implemented stringent kernel access restrictions, leading the malware authors to shift their focus to developing userland malware. This evolution in tactics necessitated a corresponding shift in forensic research toward analyzing userland runtime environments. While significant memory analysis capabilities have been developed for various runtime environments, including Android, Objective-C, and.NET, no effort has addressed the analysis of Python despite its growing popularity among legitimate software developers and malware authors. To address this critical gap, we present a comprehensive analysis of the Python runtime, encompassing its hierarchical memory management, garbage collection mechanism, and thread execution context management. We automated this analysis by developing a suite of new Volatility 3 plugins that provide detailed visibility into Python applications, including classes and their runtime instances, modules, functions, dynamically generated values, and execution traces across application threads. We evaluated our plugins against real-world malware samples, including cryptocurrency hijackers, ransomware variants, and remote access trojans (RATs). Results demonstrated 100% extraction accuracy of application objects within practical time constraints. The plugins recovered critical artifacts, including cryptocurrency wallet addresses, encryption keys, malicious functions, and execution paths. Through these new automated analysis capabilities, investigators of all levels of experience will be able to detect and analyze Python-based malware.

1. Introduction

In the realm of digital forensics and incident response (DFIR), memory forensics has emerged as an essential investigative approach, particularly in combating modern malware that operates solely in system memory. Such malware avoids leaving artifacts on disk while also utilizing encrypted network communications, making traditional storage-based and network forensic techniques insufficient. The significance of memory forensics is highlighted by the recent requirement of the Cybersecurity and Infrastructure Security Agency (CISA) for memory analysis when responding to incidents. This requirement started after two severe incidents in 2021, the SolarWinds supply chain attack and the widespread exploitation of zero-day vulnerabilities in Microsoft Exchange, both of which relied heavily on memory-only payloads (Williams, Crowe, CISA, a,b).

Memory forensics enables a comprehensive reconstruction of the system state at acquisition time. However, prior research has primarily focused on analyzing kernel-level data structures for malware detection since kernel-level malware often employs sophisticated evasion techniques to bypass live and disk forensics. The power of this malware eventually forced major operating system vendors to implement rigorous security measures restricting kernel access. As a result, malware authors shifted their focus toward userland processes, exploiting system APIs to perform various malicious activities, including hardware monitoring, credential access, and lateral movement. This shift in malware tactics prompted researchers to expand their investigation into various userland runtime environments, including Android (Case, 2011), Objective-C and Swift (Manna et al., 2021), .NET (Manna et al., 2022), and JavaScript (Wang et al., 2022). Despite significant memory analysis capabilities developed for these environments, current research lacks analysis of the Python runtime, despite its growing popularity among legitimate software developers and malware authors.

* Corresponding author.
E-mail addresses: alih16@vcu.edu (H. Ali), andrew@dfir.org (A. Case), iahmed3@vcu.edu (I. Ahmed).

https://doi.org/10.1016/j.fsidi.2025.301920

Python's versatility, extensive library ecosystem, and cross-platform compatibility have made it increasingly attractive to malware authors. Its interpreted nature and dynamic runtime environment facilitate sophisticated malware operations, including code injection, obfuscation, dynamic execution, and runtime modification of system behaviors. Recent security incidents have demonstrated diverse exploitation of these capabilities. *PyLoose* leveraged fileless execution techniques to deploy cryptominers in cloud environments (News). The *Connecio* information stealer masqueraded as a legitimate CrowdStrike Falcon update (CrowdStrike), while *PureHVNC* remote access trojan exploited multi-stage Python-based loaders and hidden virtual network computing to establish covert control of Windows systems while evading detection (Wan).

The threat landscape has expanded to include supply chain attacks targeting third-party Python packages (Nguyen et al., 2024; Li et al., 2023). Notable examples include the *PondRAT* backdoor, which enabled unauthorized access to the compromised systems (Nelson, 2024), *fshec2* package, which leveraged bytecode compilation to obscure malicious activities (Nelson, 2023b), and *BlazeStealer* package, which masqueraded as legitimate code obfuscation tools, targeting developers to gain comprehensive control of their systems through Discord-based command and control (Schwartz). Malware authors have also exploited packing tools, such as *PyInstaller,* to generate compiled bytecode files, effectively evading detection by antivirus engines (Koutsokostas and Patsakis, 2021). Moreover, specialized malware has emerged targeting specific platforms, such as ransomware designed for Jupyter Notebook environments (Morag), *NodeStealer* attacking Facebook Ads Manager (Aira Marcelo), and *Androxgh0st* focusing on exfiltrating credentials from *Laravel* Framework applications (FBI and CISA).

This paper documents our efforts to address the significant gap in the current memory forensics literature. We provide the first comprehensive, structured analysis of the Python runtime environment. Our novelty lies in the cross-platform analysis of Python's hierarchical memory management, garbage collection mechanism, and thread execution context management. We conduct this analysis at multiple hierarchical levels, starting with locating the main Python application class and extracting its key attributes (i.e., name and file path). Through this entry point, we access all the other active components of the application, including classes and their runtime instances, variables, modules, and functions. For the modules within each class scope, we identify their name, parent package, file path, and initialization state. Similarly, by analyzing the data structure of the functions, we determine their name, parent module, and file location. Further analysis of the function code reveals the parameter count, names of local variables, and compiled bytecode. To understand the application runtime behavior, we reconstruct the execution paths by iterating the function call stacks and their frame objects, recovering the runtime values of local variables within their respective execution contexts. We implemented these capabilities as a suite of new Volatility 3 plugins (The Volatility Foundation, 2017), which enable automated and reproducible analysis of Python applications.

We evaluated our plugin suite against a diverse set of real-world Python-based malware samples executed from source code and when packaged by PyInstaller in both Linux and Windows environments, using extraction accuracy, execution efficiency, and recovery of malicious artifacts criteria. The *Py_Class* plugin demonstrated 100% accuracy in extracting all application objects (i.e., modules, functions, classes, and variables), while all plugins exhibited practical execution times of ~3–19 seconds, depending on plugin implementation and application complexity. The plugins also proved their effectiveness in identifying malicious classes and the criminal's wallet address within the BTC-Clipper, recovering the encryption keys generated dynamically by the ransomware, reconstructing GonnaCry's execution traces, and identifying the malicious activities of PythonRAT, such as keylogging and web camera capturing.

Through these new automated analysis capabilities, we not only enable investigators to detect and analyze Python-based malware but also establish a foundation for standardized Python memory forensics methodology. Our approach aligns with existing methodologies for other runtime environments while recovering key artifacts essential for malware analysis that directly reveal malware's purpose and capabilities. This systematic framework enables investigators to perform in-depth Python malware analysis without traditional reverse engineering techniques and can be further extended by the forensic community.

2. Related work

The abuse of userland runtimes by malware led researchers to focus on a structured analysis of such runtimes to identify malicious behaviors. For instance, Android runtime has been a significant target for malware, resulting in severe consequences, such as accessing call history, text messages, microphones, and cameras (Smmarwar et al., 2024). To detect these activities, many researchers worked on analyzing this runtime (Sylve et al., 2012; Case and Richard III, 2017; Ali-Gombe et al., 2020, 2019; Tam et al., 2015). Similarly, the Objective-C and Swift runtimes of macOS have been targeted by malware. Therefore, Case and Richard (Case and Richard III, 2016) developed Volatility plugins for analyzing these runtimes and detecting malicious activities, including keystroke logging, suspicious method sizzling, and runtime manipulation. Manna et al. (2021) created plugins to enumerate loaded classes, locate class instances, parse instance variables, decode variable types, and examine class methods to detect suspicious API usage and identify malicious function calls. The same author also created plugins to analyze the.NET and.NET Core runtimes (Manna et al., 2022). These plugins could detect and extract memory-resident assemblies while analyzing runtime components, including classes, fields, methods, and native code imports, to identify suspicious behavior. Wang et al. (2022) focused on analyzing the V8 JavaScript runtime environment within Node. js's V8 engine by developing plugins to extract and trace V8 objects using the MetaMap structure. However, our extensive review of existing forensic tools and literature confirmed that no current approaches provide comparable capabilities for Python runtime analysis, highlighting the significant gap our work addresses.

3. Memory analysis methodology

Fig. 1 illustrates our memory analysis methodology of the Python runtime environment. Unlike simple bit-by-bit parsing, our approach models Python's complex memory architecture and implements analysis capabilities that extract and interpret runtime artifacts. By systematically mapping relationships between memory components, we enable the reconstruction of execution traces and behavioral patterns, revealing malicious activities that would remain hidden to isolated object analysis. The Python interpreter is primarily implemented through CPython, the reference implementation written in C that enables direct interfacing with C code through CPython extension modules written in C/C++. At the core of this environment, the `_PyRuntime` structure maintains critical interpreter state information. As a global variable of type `_PyRuntimeState`, it manages essential runtime parameters, including the main interpreter initialization state, garbage collection mechanism, thread management mechanism, and global configuration attributes.

Locating `_PyRuntime` requires different approaches between operating systems due to their distinct executable formats. In Linux environments, the relative virtual address (RVA) of `_PyRuntime` is accessed through the dynamic symbol table (`.dynsym` section) of the ELF header (Oygenblik et al., 2024). While in Windows environments, `_PyRuntime` is located in the Export table of the system-installed Python module (.dll). After locating `_PyRuntime`, we use its management mechanisms to analyze Python applications comprehensively. This analysis encompasses two main components: *Object Retrieval* and *Frame Chain Retrieval*. These components enable us to extract Python objects

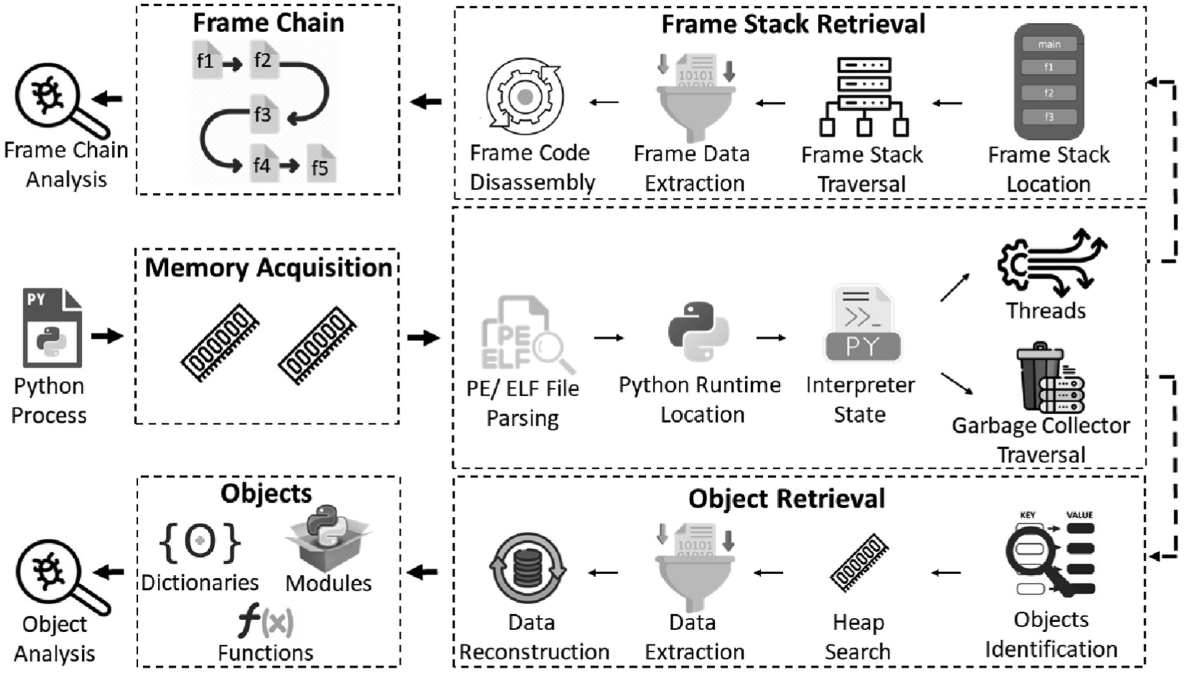

Fig. 1. Overview of the memory analysis methodology of the python runtime environment.

from memory, iterate through function call stacks and their frames to reconstruct execution paths, and recover runtime values of local variables within their respective execution contexts.

3.1. Objects retrival

The first step toward extracting Python objects from memory is to locate the Garbage Collector (GC) of the Python interpreter. It manages the lifecycle of Python objects within the process scope through a three-generation scheme, as illustrated in Fig. 2. Generation 0 contains newly created objects, Generation 1 contains objects that survive Generation 0 collections, and Generation 2 maintains long-lived objects. Each generation maintains a linked list of `PyGC_Head` structures, along with object count tracking and collection thresholds controlling the trigger of collection cycles (Foundation, b).

The `PyGC_Head` structure, which varies in size by system architecture (12 bytes for 32-bit and 24 bytes for 64-bit systems), contains `uintptr_t _gc_next` and `_gc_prev` pointers that form a circular doubly-linked list. This structure precedes every Python object tracked by the GC, with the actual object data starting 24 bytes after the `PyGC_Head`. Each object begins with a `PyObject` header containing an `ob_type` field that points to a `PyTypeObject` structure, defining the object's type.

The traversal of the garbage collector aims to identify the active

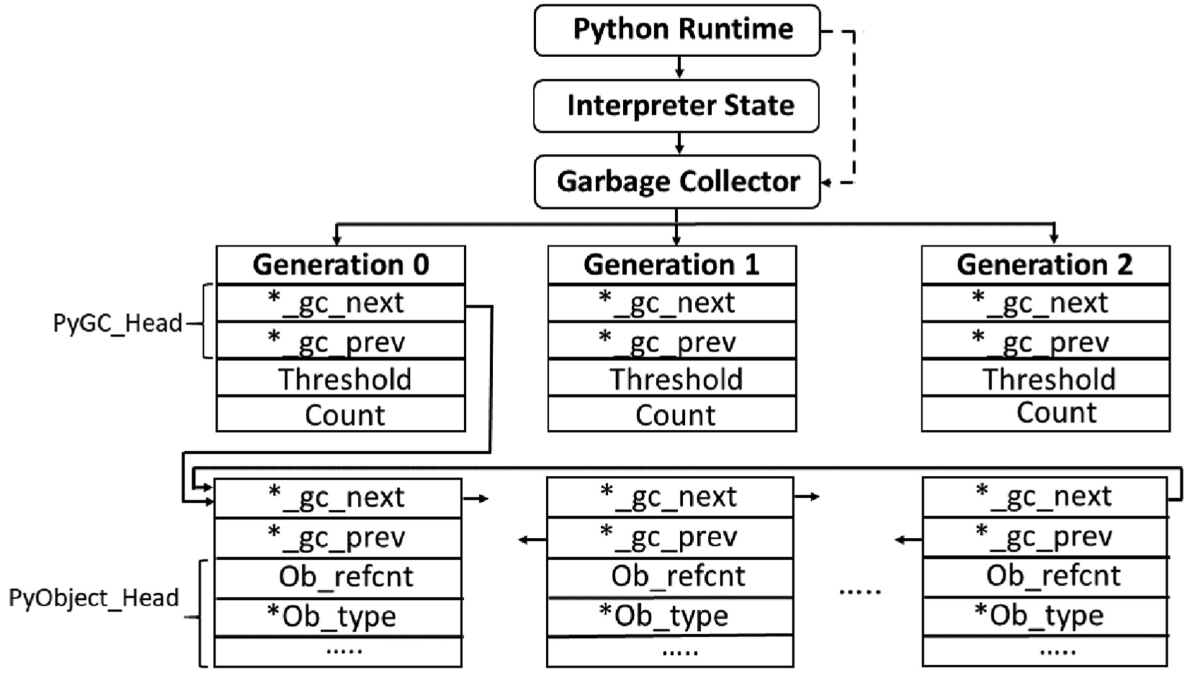

Fig. 2. Python garbage collector hierarchy.

Python objects in the heap memory, from which we extract raw data and reconstruct each object's structure through memory layout interpretation. The process begins with locating the root module (`sys`) (Sourcerer.io), a fundamental component that persists throughout the life of the application and is consequently placed directly in Generation 2 during the interpreter initialization. Represented as a `PyModuleObject` structure, `sys` stores a comprehensive registry of the application modules through its `md_dict` dictionary. This dictionary is particularly significant as it encompasses references to all active modules, most notably the `__main__` module that encapsulates the main file of the Python application. From this file, we expand our analysis to cover all other files used and imported by this file.

Building upon this hierarchical structure, accessing the application objects requires analysis of the `PyModuleObject` structure of the main file, which is leveraged by the following plugins. *Py_Class* first inspects the main file (entry point) of the application and all of its classes, while *Py_Module*, *Py_Function*, and *Py_Code* expand the analysis to its modules, functions, and code objects, respectively. It is important to note that while the garbage collector's fundamental structure remains consistent across Python versions, its location varies. For example, in Python 3.8, the garbage collector is directly accessible through the `_PyRuntime` structure, whereas Python 3.11 accesses it through the main interpreter, which is a field within the `_PyRuntime` structure. Our plugins adapt to the version present while performing analysis.

3.1.1. Py_Class Plugin

This plugin focuses on analyzing the classes of the Python application and providing detailed information about their attributes, objects, and instances. It starts with the main file that represents the main entry of the application by analyzing its `PyModuleObject` data structure. This structure maintains references to all of the main file components, including built-in attributes (e.g., `__name__`, `__doc__`, `__file__`), classes, imported modules, functions, variables, namespaces, method definitions, and decorators.

Python represents the classes as *type* objects with a `PyTypeObject` structure. This structure facilitates class identification through its `tp_name` field and contains a `tp_dict` pointer that references a `PyDictObject` structure, which maintains the runtime state of the class through its attributes and objects. On the other hand, each class instance has a type name matching its class name and is represented in memory as a `PyDictObject` data structure that maintains the instance's state attributes and values. While these values are accessible within the class's visibility scope, local variables within class functions remain inaccessible as they exist only within the function's local scope.

For each object within the main file and each of its classes, *Py_Class* outputs the process ID along with the object's type, name, memory address, and associated value. Given that modules and functions are analyzed by subsequent plugins, *Py_Class* provides only their names and addresses as values. Moreover, due to the lazy loading mechanism of Python, classes that remain unused or unimported by the main class are absent from memory and thus excluded from the plugin's analysis scope.

3.1.2. Py_Module Plugin

This plugin analyzes the `PyModuleObject` structure of the modules imported by the main file and all of its classes. This fundamental structure encapsulates fields essential for malware analysis, such as `md_name` that defines the module's runtime name, `md_dict` that points to the module's dictionary of attributes, functions, and variables, and textttmd_def which links to the module's `PyModuleDef` structure. Within `PyModuleDef`, essential fields encompass `m_name` for the original module name, `m_doc` for documentation strings, and `m_methods`, which holds an array of `PyMethodDef` instances containing all a module's functions. It is important to note that the `PyModuleDef` structure represents the C-level module definition, and it is primarily associated with built-in modules written in C like `sys`, `time`, and `builtins`.

Although Python uses the `PyModuleObject` structure to represent all the modules, their underlying dictionaries vary in content depending on each module's specific purpose. Our plugin examines the core attributes commonly present across these module dictionaries. These essential attributes reside in `md_dict` and hold the module's name (`__name__`), parent package (`__package__`), and file path (`__loader__`). Additionally, the `__spec__` dictionary, through its `_set_fileattr` field, indicates whether the module possesses `__file__` and `__cached__` attributes, while `_initializing` reveals whether the module is currently being imported or has completed initialization. Given that the built-in modules are compiled directly into the Python runtime instead of existing as separate files on the file system, their `_set_fileattr` attribute is always False. As output, for each module, *Py_Module* identifies the name, associated package, file paths, and initialization status at memory capture time. Through this functionality, the plugin enables the detection of suspicious modules, particularly those dynamically loaded during runtime, and identifies their file paths.

3.1.3. Py_Function Plugin

This plugin also extends the capabilities of *Py_Class* by analyzing Python functions through their `PyFunctionObject` data structure. This structure maintains consistent fields across Python versions, making it particularly valuable for malware analysis through its comprehensive representation of function attributes and behaviors.

At its core, *Py_Function* examines four key fields of the `PyFunctionObject` structure, including `func_name` to identify the function name, `func_globals` to access the dictionary of global objects accessible by the function, including the `__file__` attribute that specifies its source file location, `func_module` to determine the parent module of the function, and `func_code` to point to the function's associated code object. This code object contains the function's implementation details and is analyzed by the following *Py_Code* Plugin. Through this analysis, *Py_Function* constructs a comprehensive profile of each function, including its memory address, name, hosting module's name, and source file location, which are the essential components for tracking and investigating suspicious functions within Python applications.

3.1.4. Py_Code Plugin

Py_Code provides detailed insights into the code of a Python function. While executing, the Python interpreter compiles the source code into bytecode - a low-level instruction set optimized for execution by the Python Virtual Machine (PVM). This bytecode comprises a sequence of instructions. Each instruction consists of a single byte operation (opcode) potentially accompanied by additional arguments, directing the PVM to perform specific operations, such as loading values, performing arithmetic, and controlling the program flow.

Py_Code leverages the `PyCodeObject` structure that encapsulates the executable code at runtime. This structure contains not only the bytecode but also its comprehensive metadata, such as the constant values and variable names. By analyzing the key fields of this structure, *Py_Code* facilitates the extraction of critical code information. Such fields include `co_name` to identify the associated function name, `co_argcount` to reveal the number of positional arguments, `co_nlocals` to indicate the number of local variables, `co_varnames` to list the names of local variables, and `co_code` to extract the actual bytecode instructions. As bytecode itself is not human-interpretable, *Py_Code* performs bytecode disassembly to generate a readable sequence of operations for each function. This disassembled representation enables a detailed analysis of execution logic and control flow, facilitating the detection of malicious code.

It is crucial to note that this plugin provides the static disassembled instructions of the function's bytecode, such as `LOAD_GLOBAL` or `LOAD_METHOD` for loading functions, `STORE_FAST` for storing variables, and `LOAD_FAST` for accessing variables. However, these instructions provide an abstract view of the code without exposing the actual values processed during runtime. Therefore, to gain visibility into

the actual runtime context of functions, including their execution state and processed values, it is necessary to analyze the function call stack, a task facilitated by our *Py_Stack* plugin.

3.2. Frame chain retrival

Python applications leverage multi-threading for parallel execution paths, with each thread encapsulated by a `PyThreadState` structure and maintaining its own function call stack. When a Python application executes, each function call generates a new frame object added to the current thread's frames. Therefore, we call the function call stack as *Frame Stack*. The `PyThreadState` structure maintains a pointer to the current execution frame, while each frame is represented by the `PyFrameObject` structure and contains a back pointer to its predecessor frame, creating a chain of frames, as shown in Fig. 3. These frame structures serve as runtime representations of function objects, encapsulating local and global variable namespaces, execution state information, and references to their corresponding code objects (`PyCodeObject`) (Foundation, a). Our *Py_Stack* plugin locates these frame stacks within threads, traverses them using the back pointers, and analyzes each frame's structure to reconstruct the execution paths and recover their local variables.

3.2.1. Py_Stack plugin

Unlike the above-mentioned GC-based plugins, *Py_Stack* focuses on capturing the runtime state of the Python application at the moment of memory acquisition. It uniquely traces function call sequences and execution paths, making it particularly valuable for identifying root causes of malicious behavior and accessing actual values of variables processed at runtime. The plugin begins by iterating through the application threads, accessing each thread's frame stack, and then analyzing the `PyFrameObject` structure associated with each of its frames. This structure contains essential fields such as `f_code` which points to the frame's bytecode, `f_globals` that represent a dictionary of the modules and global variables within the frame's scope, and `f_locals` which is a dictionary created only when code explicitly requests local variables (e.g., via `locals()`) or when debugging tools introspect the frame. In normal execution, local variables are kept in a more efficient array field, called `f_localsplus`. Moreover, `PyFrameObject` also includes a `f_back` field that points to the previous frame, linking all active frames into a frame chain.

Python 3.11 introduced architectural changes to this structure to optimize frame handling. The structure was renamed to `_PyInterpreterFrame` while maintaining its core functionalities. Additionally, the `f_back` field was renamed to `previous` while preserving other field names and purposes. `_PyInterpreterFrame` now maintains a separate pointer to `PyFrameObject`, which is allocated on demand specifically for introspection, tracing, or debugging operations. This new `PyFrameObject` contains only essential fields related to debugging, such as `f_lineno` for the code's location within its source file and `f_trace_lines` for enabling line-by-line tracing capabilities. This architectural separation in Python 3.11 optimizes memory usage by grouping frequently accessed fields within `_PyInterpreterFrame` while isolating specialized introspection fields in `PyFrameObject` (Foundation, d). To recover all possible details of code execution, we implemented *Py_Stack* as two complementary Volatility plugins, *Py_Stack_Call* and *Py_Stack_Var*, that leverage `PyFrameObject` to facilitate comprehensive runtime analysis of Python applications. ***Py_Stack_Call***. This plugin reconstructs execution paths by leveraging the metadata fields of the frames, including `f_code` to access bytecode information (i.e., `co_code`, `co_filename`, `co_name`), `f_back` for frame chain traversal, and `f_lineno` for precisely locating the frame's code within its source file. Through these fields, *Py_Stack_Call* provides a detailed execution timeline for each thread, with chain elements formatted as "code_name (file_name: line_number)". Moreover, when executed with the `–dump` argument, the plugin generates a separate file for each thread containing the disassembled bytecode of its frames. ***Py_Stack_Var***. This plugin complements the functionality of *Py_Stack_Call* by extracting the

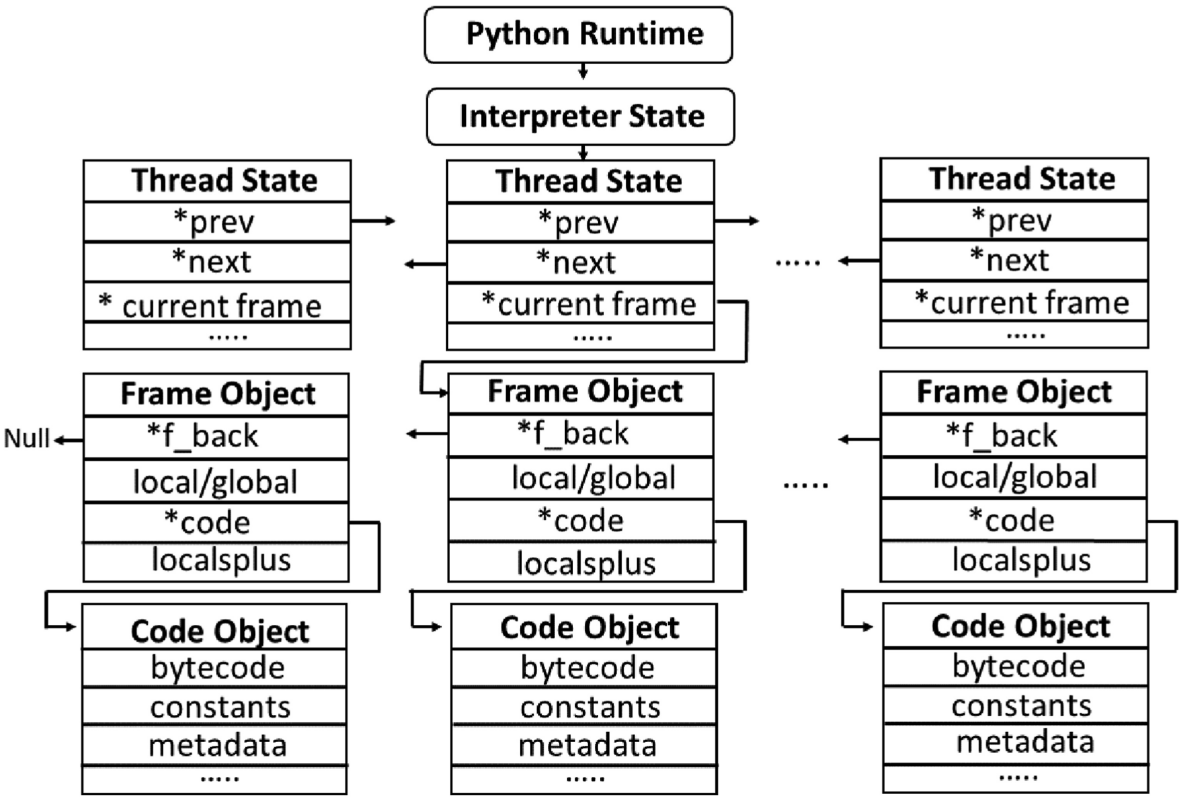

Fig. 3. Python thread states hierarchy.

runtime values of the frame's local variables. To achieve this, the plugin analyzes the `localsplus` array in the `PyFrameObject` structure, which serves as the primary storage mechanism for live values during execution. The first part of `f_localsplus`, from index 0 to `co_nlocals-1`, stores pointers to the actual values of local variables, where each slot corresponds to a name defined in `co_varnames` with the same order. In contrast, the subsequent part of this array stores pointers to temporary values and intermediate results during execution. Thus, the total size of `f_localsplus` in bytes equals (`co_nlocals` + `co_stacksize`) multiplied by the size of each pointer slot.

4. Experimental evaluation

We evaluated our Volatility 3 plugins against a diverse set of real-world Python-based malware samples, including BTC-Clipper (NightfallGT), two distinct ransomware implementations (Python-Ransomware (ncorbuk) and GonnaCry (tarcisio marinho)), and a remote access trojan (PythonRAT) (safesploit). Our evaluation utilized open-source samples to provide ground truth baselines for accuracy verification, represent diverse malware categories that employ techniques commonly found in closed-source variants, and finally, using documented samples enhances reproducibility, allowing other researchers to validate our findings. While closed-source malware may employ additional obfuscation techniques, the core Python runtime structures remain consistent regardless of source availability. For instance, the dictionary associated with the `PyTypeObject`, `PyDictObject`, and `PyModuleObject` structures contains keys representing the names of their objects and values referencing the implementations, making our approach equally effective for unknown malware. Moreover, the demonstrated 100% extraction accuracy across diverse samples, including those packaged with PyInstaller, validates our plugins' effectiveness for analyzing both open and closed-source malware.

The BTC-Clipper and Python-Ransomware are designed for Windows systems, while GonnaCry specifically targets Linux environments, and PythonRAT is compatible with both platforms. For each malware, we created a separate virtual machine (VM) using VMware Workstation 16 Pro (version 16.2.5). These VMs were configured with either Windows 10 or Ubuntu 20.04.6 LTS as the operating system, 4 GB of RAM, and Python 3.8.10 (Foundation, c). For installation, we followed the official GitHub documentation. Moreover, for PythonRAT and GonnaCry, which require client-server architecture, we deployed their respective server components within the same virtual machine environment. We ran the plugins on a VM equipped with Ubuntu 20.04.6 LTS and Volatility 3 (version 2.5.0). While Volatility 3 includes a prebuilt Windows 10 symbol table, we generated kernel debugging symbol tables for Linux kernels. Analyzing Python objects also requires Python-specific symbol tables for both operating systems. Therefore, we used the *dwarf2json* tool (Volatility Foundation) to generate the debugging symbol tables for the Linux kernels and Python interpreter.

4.1. Evaluation criteria

We evaluated our plugins using three key criteria:

- **Extraction Accuracy:** Measures the plugin's ability to recover Python objects from memory against a ground truth baseline established through static source code analysis using Python's Abstract Syntax Tree (AST). It is calculated as: $\text{Ext\_Acc} = \min\left(\frac{N_{\text{found}}}{N_{\text{total}}}, 1.0\right) \times 100\%$; N_{found} and N_{total} represent the number of recovered and baseline objects, respectively.
- **Execution Efficiency**: Measures the processing time required by each plugin to extract and analyze objects from memory, which varies based on plugin implementation complexity and malware characteristics.
- **Recovery of Malicious Artifacts**: Evaluates the plugin's ability to recover critical forensic artifacts, including encryption keys, cryptocurrency wallet addresses, C2 connections, and malicious execution traces.

Table 1 shows the *Py_Class*'s extraction accuracy, providing a comprehensive visibility into all application objects and establishing the foundation upon which the other plugins are built. Extracted objects typically exceed the static analysis baseline because *Py_Class* captures explicitly defined components and dynamically loaded objects that static analysis cannot detect. These include classes and their instances states, modules used by both the interpreter and application, dynamically generated values, and defined and imported functions by all modules, providing a complete representation of the Python environment. Table 2 shows the average execution time of the plugins across the malware samples. *Py_Module* shows the highest execution times due to its recursive traversal of module hierarchies and extensive metadata extraction, while *Py_Code* performs most efficiently, focusing on bytecode analysis. Execution time is influenced by both the plugin complexity and the memory traversal path—from generation 2 (where *sys* module resides) through generations 0 and 1. Object count significantly impacts performance, as seen with *PythonRAT* (7681 objects) having the longest processing times, while *GonnaCry* shows that object structure can be more influential than count, with fewer objects but not always faster processing.

4.2. BTC-clipper

BTC-Clipper, also known as a "Bitcoin Clipper", monitors the clipboard for cryptocurrency wallet addresses and replaces them with attacker-controlled addresses to hijack transactions. In this scenario, recovering the attacker's wallet address is crucial for tracking transactions and fund movement, thus helping in criminal investigations and linking to other potential incidents.

The output of the *Py_Class* plugin, as shown in Fig. 4, highlights key details of the main module in BTC-Clipper (*btcClip.py*). The criminal's wallet address and the destruct message are stored as global variables with fixed string values. The output also shows the class attributes, including `__name__`, `__package__`, and `__loader__`. The None value of `__package__` indicates that the main file does not belong to a specific package, while `__loader__` stores the relative file path of the class due to executing the class directly from its local directory. The output further identifies two classes in the main file, `Clipboard` and `Methods`, represented as `PyTypeObject` types. For each class, the plugin displays its attributes and associated components, including functions, static methods, and variables.

4.3. Python-Ransomware

This repository implements a simple ransomware variant that combines asymmetric and symmetric encryption. The *RSA_private_public_keys.py* class generates the RSA key pair, while *RansomWare.py* handles the core encryption using the `Fernet` module for AES-CBC encryption with HMAC authentication. A Fernet symmetric key is first generated and used to encrypt files in the "*localRoot*" directory through a *crypter* object. This key is then encrypted with the RSA public key, and the *crypter* reference is set to `None`. The malware architecture comprises three key components: the main file, an instance of *Ransomware* class executing core functions, and an instance of *Fernet* class managing cryptographic operations. The *Py_Class* plugin successfully recovered the 16-byte HMAC authentication key and 16-byte AES encryption key from the Fernet instance, along with other sensitive data such as filesystem paths and the victim's public IP. Detailed output of this analysis is provided in Appendix A.

Table 1
Extraction accuracy of the *Py_Class* plugin across the malware samples.

Malware	Ground Truth Objects				Extracted Objects				Ext_Acc
	Module	Func.	Class	Var.	Module	Func.	Class	Var.	
BTC-Clipper	8	10	2	4	11	10	2	4	100%
Python-Ransomware	15	13	1	1	15	22	2	1	100%
GonnaCry	19	5	0	0	21	6	0	0	100%
PythonRAT	14	13	0	1	14	14	0	1	100%

Table 2
Extraction time of the Plugins across the Malware Samples.

Malware	Avg. Extraction Time (Second)			
	Py_Class	Py_Module	Py_Function	Py_Code
BTCClipper (Windows)	4.8605	7.0493	4.1129	2.6000
Python-Ransomware (Windows)	4.9797	12.1914	6.5897	4.2669
GonnaCry (Linux)	3.2376	16.0314	5.2902	3.2136
PythonRAT (Linux)	8.4064	19.5843	15.3146	8.0046

4.4. GonnaCry ransomware

GonnaCry implements more sophisticated key protection mechanisms that present unique forensic challenges. The ransomware employs a multi-layered encryption strategy that generates a victim-specific RSA key pair and encrypts each target file in the victim's home directory and desktop with a unique AES key, which is subsequently encrypted using the public RSA key. To prevent decryption of these AES keys, *GonnaCry* encrypts the RSA private key with the server-side public key, which is stored in the *variables. py* module. Thus, only the attacker with access to the server-side private key can decrypt the victim's private RSA key and subsequently recover the file-specific AES keys. Furthermore, this ransomware implements memory cleanup by immediately nullifying the private key, explicitly deleting it, and forcing garbage collection to prevent forensic recovery. In addition to this challenge, all keys are generated locally within the class's functions, making *Py_Stack* the only plugin capable of key recovery.

Our experiment was carried out with two targeted files, *file1. txt* and *file2. txt*, containing "*Hello from file1 !!!*" and "*H. from file2 !!!*", respectively. Memory was acquired at three critical points: during the encryption of the first file (*Snapshot*$_1$), during the encryption of the second file (*Snapshot*$_2$), and after completing the encryption process by the main function, `menu()`, (*Snapshot*$_3$). Due to the arbitrariness of `os.walk()` used by the malware, *file2. txt* was encrypted before *file1. txt*. Table 3 illustrates the output of *Py_Stack_Call* with these three snapshots. It revealed a single-threaded execution path with three frames during file encryption. However, when `start_encryption()` completes and returns its encoded keys and filenames, its frame is removed from the stack, leaving only the `menu()` frame and the main frame of the module as shown with *Snapshot*$_3$. This multi-snapshot analysis demonstrates our framework's ability to track malware behavior over time, revealing how encryption keys and artifacts evolve throughout the malware's lifecycle—a capability essential for comprehensive forensic investigation.

Fig. 5 demonstrates the output of *Py_Stack_Var* against the `start_encryption` frame during encryption of the second file, which includes the recovery of the AES key used for this file and its original plaintext content. Since ransomware typically targets numerous files on the system, the complete encryption process takes a significant execution time. This time window provides investigators with a crucial forensic opportunity, as memory acquisition during the encryption

Table 3
Output of the *Py_Stack_Call* plugin during the *GonnaCry*'s lifecycle.

Mem. Capture Point	No. Thread	No. Frames	Frame Chain
Snapshot$_1$	1	3	<module >(main.py:1) → menu (main. py:75) → start_encryption (main.py:48)
Snapshot$_2$	1	3	<module >(main.py:1) → menu (main. py:75) → start_encryption (main.py:48)
Snapshot$_3$	1	2	<module >(main.py:1) → menu (main. py:75)

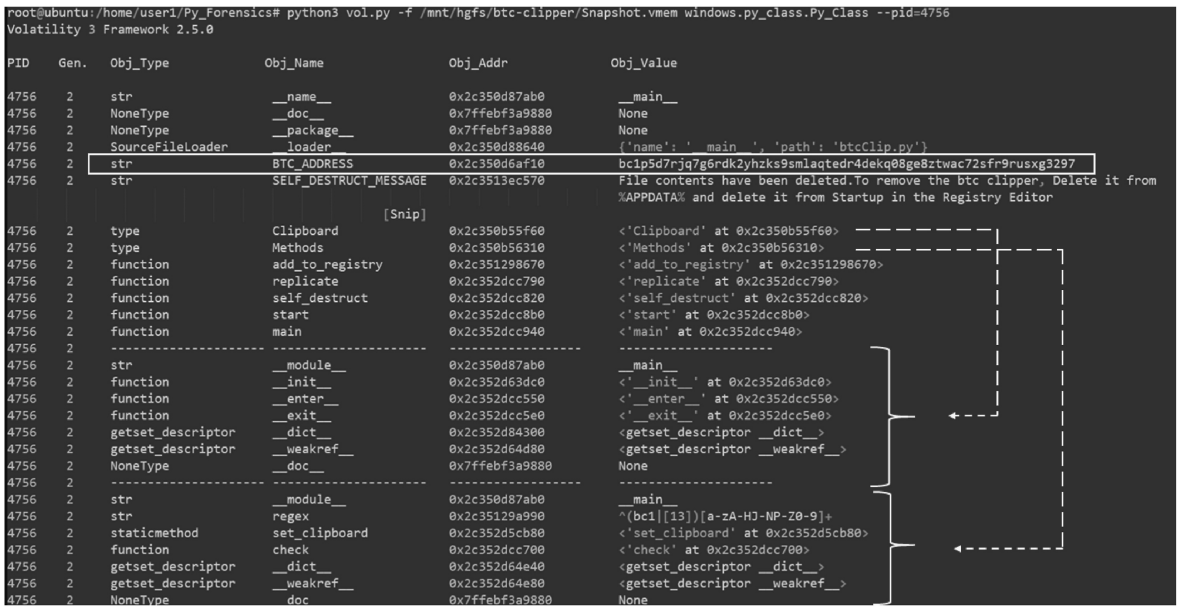

Fig. 4. Output of the *Py_Class* plugin against *BTC-Clipper*.

```
root@ubuntu:/home/user1/Py_Forensics# python3 vol.py -f /mnt/hgfs/gonnacry/Snapshot2.vmem linux.py_stack_var.Py_Stack_Var --pid=3068 --target-frame=start_encryption
Volatility 3 Framework 2.5.0

PID     Function Name       Variable Name               Variable Value                                                    Previous Key

3068    start_encryption    files                       [b'L2hvbWUvaGFsYS90ZXN0L2ZpbGUyLnR4dA==', b'L2hvbWUvaGFsYS90ZXN0L2ZpbGUxLnR4dA==']
3068    start_encryption    AES_and_base64_path         [[b'RnnCzowoW0G6wMO/jZsHcGWTSzkRCz3Ca8Z4ufste2bba5v/N37W6QLORBAohAoMoD/Fw/XYnPSjHQ5'\
                                                          'rxyffrwDbMMbUzq5tZanAamQCZ0tR+7K4H3ewHRcmO1QYe9zTcmeYX6Ks3revIiLQFKPHgTjvS5UbiANu9XtwqW7x6Bw=',
                                                        b'L2hvbWUvaGFsYS90ZXN0L2ZpbGUyLnR4dC5HTk5DUlk='],
                                                       [b'nchfD5TQEXn5WTv71dLkOfLKfPYbhuCudM/aP12HSEKvYsE4+lhQjTTTvMcs6QdjiNXQJcrGTyHjlklZ20ezI'\
                                                          '7RUFKH2Ho2K3W3Qae1oYbx/WI9Qny5HpJ3reikijtrWOkKRef7pn3uzcDmFZKgnx8t5IW5FDWBRRpXC8/DHehw=',
                                                        b'L2hvbWUvaGFsYS90ZXN0L2ZpbGUxLnR4dC5HTk5DUlk='']]
3068    start_encryption    found_file                  b'L2hvbWUvaGFsYS90ZXN0L2ZpbGUxLnR4dA=='
3068    start_encryption    key                         b'nchfD5TQEXn5WTv71dLkOfLKfPYbhuCudM/aP12HSEKvYsE4+lhQjTTTvMcs6QdjiNXQJcrGTyHjlklZ20ezI'\
                                                          '7RUFKH2Ho2K3W3Qae1oYbx/WI9Qny5HpJ3reikijtrWOkKRef7pn3uzcDmFZKgnx8t5IW5FDWBRRpXC8/DHehw='
3068    start_encryption    AES_obj                     {'bs': 32, 'key': b'\xfd\xfdo\xc0\xacuP\x82\xd4\xbd^\x11\x9d@\xed\xc3\xaeo3n\x81eX+\x8f\x1d"\x1d@=\x8d2'}
3068    start_encryption    decoded_path                b'/home/user1/test/file1.txt'
3068    start_encryption    f                           'CPython Extension'
3068    start_encryption    file_content                b'Hello from file1 !!!'
3068    start_encryption    e                           None
3068    start_encryption    encrypted                   b'+8Wxvv1JtthBExXxjZHqoPP96rRQ2yEuLWhf4dmt3FOOian9vwM/M7s1MNcc6wJZ'
3068    start_encryption    new_file_name               /home/user1/test/file1.txt.GNNCRY
3068    start_encryption    base64_new_file_name        b'L2hvbWUvaGFsYS90ZXN0L2ZpbGUxLnR4dC5HTk5DUlk='
```

Fig. 5. Output of the *Py_Stack_Var* Plugin against the *GonnaCry*'s *start_encryption* Frame (*Snapshot*$_2$).

process of any file, including the final one, facilitates the recovery of all previously generated encryption keys. Analysis of the second memory snapshot validates this finding, as shown by the *Py_Stack_Var* output. Specifically, the `AES_and_base64_path` list maintains the encryption keys for both target files, along with their new base64-encoded filenames that have the ". GNNCRY" extension.

Fig. 6 illustrates the output of *Py_Stack_Var* with the `menu` frame from *Snapshot*$_3$. We observe that the values of `aes_keys_and_base64_path`, `enc_aes_key_and_base64_path`, `rsa_object`, and `Client_private_key` variables appear as None due to forced garbage collection by the ransomware. However, we can still recover the most recent AES key and the content of the final file. Furthermore, by base64-decoding the `line` variable, we can extract both the encryption key and the file path of the most recently encrypted file. The results of this experiment clearly show the value of our plugins while analyzing ransomware written in Python.

4.5. PythonRAT

PythonRAT implements a botnet architecture in which multiple compromised systems are controlled through a centralized Command and Control (C2) server. Its repository comprises the server-side controller (*c2. py*) and two client-side classes (*backdoor.py* and *keylogger. py*). The *backdoor. py* class serves as the primary client-side agent, establishing a connection with the C2 server and executing various commands, including file system navigation, file transfer, screenshot generation, webcam capture, and keystroke logging.

To analyze the behavior of this malware, we used the capabilities of the `Py_Module`, `Py_Function`, and `Py_Code` plugins. Fig. 7 shows the `Py_Module` output, revealing the imported modules by *backdoor. py*. Although the individual modules might appear legitimate, their combination strongly indicates malicious intent due to the presence of *socket*, *SSL*, and *requests*, which enable encrypted remote communications, *cv2* which supports screen capture and webcam monitoring, and *subprocess* which facilitates external process creation and system command execution.

Fig. 8 illustrates the suspicious functions used by both *backdoor. py*, and *keylogger. py*, including those from the `pynput.keyboard._xorg` submodule for keystroke logging. Notably, due to executing the main class (*backdoor.py*) directly from its local directory, its relative file path appears with its functions. In contrast, the full absolute path is associated with the functions of *keylogger. py*, as it is an imported module. Furthermore, libraries installed via pip, such as *pynput*, are typically located within the site-packages directory using their complete module hierarchy.

For each function, `Py_Code` displays its number of arguments and local variables besides the variable names and disassembled code. As the disassembled code is lengthy, Fig. 9 shows only the code of the `start` function while truncating the code of the `on_press` function. This output shows malicious keylogging activity. When a key is pressed, pynput's Listener automatically calls the `on_press` function with the pressed key as an argument. This function, in turn, writes the collected keys to a file to be sent to the server.

PyInstaller allows Python applications to be packaged into standalone executables that can run without requiring Python to be installed on the target system. During packaging, it compiles all Python dependencies and modules into bytecode and bundles them with the Python interpreter into a single executable. The resulting executable's

```
root@ubuntu:/home/user1/Py_Forensics# python3 vol.py -f /mnt/hgfs/gonnacry/Snapshot3.vmem linux.py_stack_var.Py_Stack_Var --pid=3068 --target-frame=menu
Volatility 3 Framework 2.5.0

PID     Function Name       Variable Name               Variable Value

3068    menu                files                       [b'L2hvbWUvaGFsYS90ZXN0L2ZpbGUyLnR4dA==', b'L2hvbWUvaGFsYS90ZXN0L2ZpbGUxLnR4dA==']
3068    menu                rsa_object                  None
3068    menu                Client_private_key          None
3068    menu                Client_public_key           b'ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABA....[TRUNCATED]'
3068    menu                encrypted_client_private_key [b'Uio\xda\xf7\xf1\xdb&\xd81MW\xbc\xbd]\xfaHH\xdeIbM\xef ....[TRUNCATED]']

                                              [Snip]
3068    menu                aes_keys_and_base64_path    None
3068    menu                enc_aes_key_and_base64_path None
3068    menu                aes_key                     b'nchfD5TQEXn5WTv71dLkOfLKfPYbhuCudM/aP12HSEKvYsE4+lhQjTTTvMcs6QdjiNXQJcrGTyHjlklZ20ezI'\
                                                          '7RUFKH2Ho2K3W3Qae1oYbx/WI9Qny5HpJ3reikijtrWOkKRef7pn3uzcDmFZKgnx8t5IW5FDWBRRpXC8/DHehw='
3068    menu                base64_path                 b'L2hvbWUvaGFsYS90ZXN0L2ZpbGUxLnR4dC5HTk5DUlk='
3068    menu                encrypted_aes_key           b' \xc6K\x89\x1eBB!C\x1d\x8a ....[TRUNCATED]'
3068    menu                encoded_key                 b'IMZLiR5CQiFDHYovyYC+HxEJ472UGvxCV9Y7jErG2LEmQMjosxh3QQeJUh5JowyplCSMDmtBXq0murip84ShCn4yeS'\
                                                          'eqP5kPrzUP1WCnxtK771Kt1AjK/UrA2QthXgeWVRtpAM+dksnUS5bvPZ3MXY6bs20vMZi27yJOrFdkf1dTBCSxcMikW2W'\
                                                          'XHo4ANPaTUBDToYUoII5/adjE2Mm+eCqlS8cpIoY1xPXdsyXS+rc+ADRcp+qd1Duuo7AGcNugNgiFFZUrU5Z+Om841ugZOk'\
                                                          'OLHcpitqbYomkxrq/DjH9jPOH5fRJq+N+rzL4NWsdkxQzwmJDNcV1Q9JQHYWnU4Q=='
3068    menu                encoded_path                b'L2hvbWUvaGFsYS90ZXN0L2ZpbGUxLnR4dC5HTk5DUlk='
3068    menu                line                        b'IMZLiR5CQiFDHYovyYC+HxEJ472UGvxCV9Y7jErG2LEmQMjosxh3QQeJUh5JowyplCSMDmtBXq0murip84ShCn4'\
                                                          'yeSeqP5kPrzUP1WCnxtK771Kt1AjK/UrA2QthXgeWVRtpAM+dksnUS5bvPZ3MXY6bs20vMZi27yJOrFdkf1dTBCSxc'\
                                                          'MikW2WXHo4ANPaTUBDToYUoII5/adjE2Mm+eCqlS8cpIoY1xPXdsyXS+rc+ADRcp+qd1Duuo7AGcNugNgiFFZUrU5Z+Om'\
                                                          '841ugZOkOLHcpitqbYomkxrq/DjH9jPOH5fRJq+N+rzL4NWsdkxQzwmJDNcV1Q9JQHYWnU4Q== L2hvbWUvaGFsYS90ZXN'\
                                                          '0L2ZpbGUxLnR4dC5HTk5DUlk=\n'
```

Fig. 6. Output of the *Py_Stack_Var* Plugin against the *GonnaCry*'s *menu* Frame (*Snapshot*$_3$).

```
root@ubuntu:/home/user1/Py_Forensics# python3 vol.py -f /mnt/hgfs/pyRAT/Snapshot.vmem linux.py_module.Py_Module --pid=57443
Volatility 3 Framework 2.5.0

PID     Module Name     Package                 Initializing            Path

57443   builtins                                Unknown                 None
57443   ctypes          ctypes                  False                   /usr/lib/python3.8/ctypes/__init__.py
57443   cv2             cv2                     False                   /home/user1/.local/lib/python3.8/site-packages/cv2/__init__.py
57443   json            json                    False                   /usr/lib/python3.8/json/__init__.py
57443   os                                      False                   /usr/lib/python3.8/os.py
57443   shutil                                  False                   /usr/lib/python3.8/shutil.py
57443   socket                                  False                   /usr/lib/python3.8/socket.py
57443   ssl                                     False                   /usr/lib/python3.8/ssl.py
57443   subprocess                              False                   /usr/lib/python3.8/subprocess.py
57443   sys                                     Unknown                 None
57443   threading                               False                   /usr/lib/python3.8/threading.py
57443   time                                    False                   None
57443   requests        requests                False                   /usr/lib/python3/dist-packages/requests/__init__.py
57443   keylogger                               False                   /home/user1/PythonRAT/client/keylogger.py
57443   ------------    -------------           --------------          --------------------
57443   os                                      False                   /usr/lib/python3.8/os.py
57443   time                                    False                   None
57443   threading                               False                   /usr/lib/python3.8/threading.py
```

Fig. 7. Output of the *Py_Module* plugin against *PythonRAT*.

```
root@ubuntu:/home/user1/Py_Forensics# python3 vol.py -f /mnt/hgfs/pyRAT/Snapshot.vmem linux.py_function.Py_Function --pid=57443
Volatility 3 Framework 2.5.0

PID     Func. Address   Func. Name              Module Name             File Name

57443   0x7fb5b0901c10  mss                     mss.factory             /home/user1/.local/lib/python3.8/site-packages/mss/factory.py
57443   0x7fb5c679d1f0  reliable_send           __main__                backdoor.py
57443   0x7fb5b0901940  reliable_recv           __main__                backdoor.py
57443   0x7fb5b05aba60  download_file           __main__                backdoor.py
57443   0x7fb5b05abaf0  upload_file             __main__                backdoor.py
57443   0x7fb5b05abb80  download_url            __main__                backdoor.py
57443   0x7fb5b05abc10  screenshot              __main__                backdoor.py
57443   0x7fb5b05abca0  get_sam_dump            __main__                backdoor.py
57443   0x7fb5b05abd30  capture_webcam          __main__                backdoor.py
57443   0x7fb5b05abdc0  persist                 __main__                backdoor.py
57443   0x7fb5b05abe50  startup_persist         __main__                backdoor.py
57443   0x7fb5b05abee0  is_admin                __main__                backdoor.py
57443   0x7fb5b05abf70  shell                   __main__                backdoor.py
57443   0x7fb5b05ac040  connection              __main__                backdoor.py
57443   --------------  --------------          ---------------         --------------
                                [Snip]
57443   0x7fb5b059eaf0  _keycode_to_keysym      pynput.keyboard._xorg   /home/user1/.local/lib/python3.8/site-packages/pynput/keyboard/_xorg.py
57443   0x7fb5b059eb80  _event_to_key           pynput.keyboard._xorg   /home/user1/.local/lib/python3.8/site-packages/pynput/keyboard/_xorg.py
57443   0x7fb5b059ec10  receive                 pynput._util            /usr/lib/python3.8/contextlib.py
57443   0x7fb5b0908040  on_press                keylogger               /home/user1/PythonRAT/client/keylogger.py
57443   0x7fb5b090fb80  read_logs               keylogger               /home/user1/PythonRAT/client/keylogger.py
57443   0x7fb5b05a1310  write_file              keylogger               /home/user1/PythonRAT/client/keylogger.py
57443   0x7fb5b05ab8b0  self_destruct           keylogger               /home/user1/PythonRAT/client/keylogger.py
57443   0x7fb5b05ab940  overwrite_file          keylogger               /home/user1/PythonRAT/client/keylogger.py
57443   0x7fb5b05ab9d0  start                   keylogger               /home/user1/PythonRAT/client/keylogger.py
```

Suspicious Functions

Fig. 8. Output of the *Py_Function* plugin against *PythonRAT*.

structure complicates analysis by antivirus engines, making PyInstaller an attractive tool for distributing Python-based malware, particularly on Windows systems. When the packaged application is executed, PyInstaller creates a parent bootloader process that extracts the Python interpreter and dependencies into a temporary directory (*_MEI**) and sets up the execution environment. A child runtime process then handles the actual execution of the application code. To evaluate the effectiveness of our analysis capabilities against PyInstaller-packaged malware, we present the ability of *Py_Class* to extract the components of a PyInstaller-packaged *PythonRAT* executable running on Windows.

As shown in Fig. 10, the `__loader__` attribute is set to `PyiFrozenEntryPointLoader` rather than `SourceFileLoader` (contrasting with Figs. 4 and 11). This loader enables *PyInstaller* to bootstrap and import compiled modules at runtime. The output also shows `_pyi_main_co`, a PyInstaller-generated component representing the compiled main application entry point, *VIRTENV* indicating PyInstaller's isolated environment, which manages dependencies, and the temporary directory path (`_MEI19882`) where the compiled modules are extracted. However, for a deeper analysis of this packed variant, other plugins can be leveraged, showing the same output of Figs. 8 and 9 besides the components of PyInstaller.

Limitations. As demonstrated with GonnaCry, malware can actively counter memory forensics by forcing garbage collection immediately after using sensitive objects, minimizing their memory residence time. Advanced malware might also implement targeted anti-forensics techniques that corrupt Python objects or manipulate memory structures to thwart analysis. Additionally, while our plugins successfully analyzed PyInstaller-packed malware, alternative packaging methods such as Py2exe, Nuitka, or Cython compilation, especially when combined with custom packers or encryptors, may pose more analysis challenges.

5. Conclusion and future work

This paper presented our efforts to analyze the memory of the Python runtime environment. We automated this analysis by developing a set of novel Volatility 3 plugins. These plugins leverage the internal structures of Python, such as traversing the garbage collector to recover the Python application objects (e.g., modules, functions, variables, and class instances) and analyzing the frame chain to reconstruct execution paths, providing unprecedented visibility into the application behavior. We demonstrated the effectiveness of our plugins through a detailed analysis of real-world Python-based malware samples, including a cryptocurrency clipper, ransomware variants, and a remote access trojan. The *Py_Class* plugin achieved 100% accuracy in extracting all application

```
root@ubuntu:/home/user1/Py_Forensics# python3 vol.py -f /mnt/hgfs/pyRAT/Snapshot.vmem linux.py_code.Py_Code --pid=57443
Volatility 3 Framework 2.5.0

PID     Func. Name      No.Args     No.Locals   Var Names       Code

57443   on_press        2           2           self, key       0: LOAD_FAST self
                                                                2: LOAD_ATTR keys
                                                                4: LOAD_METHOD append
                                                                6: LOAD_FAST key
                                                                     [...]
                                                                42: LOAD_FAST self
                                                                44: LOAD_METHOD write_file
                                                                46: LOAD_FAST self
                                                                48: LOAD_ATTR keys
                                                                50: CALL_METHOD 1
                                                                52: POP_TOP
                                                                54: BUILD_LIST 0
                                                                56: LOAD_FAST self
                                                                58: STORE_ATTR keys
                                                                60: LOAD_CONST None
                                                                62: RETURN_VALUE
57443   start   1           1           self                    0: LOAD_GLOBAL Listener
                                                                2: LOAD_FAST self
                                        Listener for Key Press Events
                                                                4: LOAD_ATTR on_press
                                                                6: LOAD_CONST ('on_press',)
                                                                8: CALL_FUNCTION_KW 1
                                                                10: SETUP_WITH 26
```

Fig. 9. Output of the *Py_Code* plugin against *PythonRAT*.

```
root@ubuntu:/home/user1/Py_Forensics# python3 vol.py -f /mnt/hgfs/pyRAT/Snapshot.vmem windows.py_class.Py_Class --pid=5056
Volatility 3 Framework 2.5.0

PID    Gen.   Obj_Type                Obj_Name              Obj_Addr            Obj_Value

5056   2      str                     __name__              0x211868910b0       __main__
5056   2      NoneType                __doc__               0x7ffd45f19880      None
5056   2      NoneType                __package__           0x7ffd45f19880      None
5056   2      PyiFrozenEntryPointLoader __loader__          0x21186a353d0       <Custom type at 0x21186a353d0>
5056   2      NoneType                __spec__              0x7ffd45f19880      None
5056   2      dict                    __annotations__       0x21186869ec0       {}
5056   2      module                  __builtins__          0x21186470310       <'builtins' at 0x21186470310>
5056   2      str                     __file__              0x21186e76ab0       C:\Users\user1\AppData\Local\Temp\_MEI19882\backdoor.py
5056   2      code                    _pyi_main_co          0x21186e9fb30       <'_pyi_main_co' at 0x21186e9fb30>
5056   2      module                  sys                   0x21186463e00       <'sys' at 0x21186463e00>
5056   2      module                  pyimod02_importers    0x21186a2e680       <'pyimod02_importers' at 0x21186a2e680>
5056   2      module                  os                    0x211869ebea0       <'os' at 0x211869ebea0>
5056   2      str                     VIRTENV               0x21186a34630       VIRTUAL_ENV
5056   2      list                    python_path           0x211869f8c40       ['C:\\Users\\user1\\AppData\\Local\\Temp\\_MEI19882\\base_library.zip',
                                                                                  'C:\\Users\\user1\\AppData\\Local\\Temp\\_MEI19882\\cv2',
                                                                                  'C:\\Users\\user1\\AppData\\Local\\Temp\\_MEI19882\\lib-dynload',
                                                                                  'C:\\Users\\user1\\AppData\\Local\\Temp\\_MEI19882']
5056   2      str                     pth                   0x2118643ccf0       C:\Users\user1\AppData\Local\Temp\_MEI19882
5056   2      module                  encodings             0x21186852a40       <'encodings' at 0x21186852a40>
5056   2      module                  pyimod03_ctypes       0x21186a2eea0       <'pyimod03_ctypes' at 0x21186a2eea0>
5056   2      module                  pyimod04_pywin32      0x21186a37310       <'pyimod04_pywin32' at 0x21186a37310>
5056   2      str                     entry                 0x21186a5f260       C:\Users\user1\AppData\Local\Temp\_MEI19882\_ssl.pyd
5056   2      module                  ctypes                0x21186a53950       <'ctypes' at 0x21186a53950>
                                                [Snip]
5056   2
5056   2      function                startup_persist       0x2118e963b80       <'startup_persist' at 0x2118e963b80>
5056   2      function                is_admin              0x2118e963c10       <'is_admin' at 0x2118e963c10>
5056   2      function                shell                 0x2118e963ca0       <'shell' at 0x2118e963ca0>
5056   2      function                connection            0x2118e963d30       <'connection' at 0x2118e963d30>
5056   2      socket                  s                     0x2118e948940       CPython Extension
```

Fig. 10. Output of the *Py_Class* plugin against *PythonRAT.exe* packed by *PyInstaller*

objects, while all plugins exhibited practical execution times of ~3–19 seconds, recovering critical artifacts, including internal malicious classes and suspicious combinations of modules and functions. The plugins could also recover dynamically generated encryption keys from ransomware samples while tracking their execution paths. In the future, we plan to develop advanced countermeasures against sophisticated Python-based malware techniques, extend our plugins to support various Python packaging methods, and introduce analysis methods to detect supply chain attacks in Python packages.

Appendix A

A Detailed Analysis of Python-Ransomware

Fig. 11 demonstrates the ability of *Py_Class* to inspect the class instances and provide their variable values. The memory was acquired immediately after calling the `ransom_note()` function of `Python-Ransomware`, the last function in the malware's execution sequence. The analysis reveals the critical components of a *Fernet* class instance, including a 16-byte `_signing_key` for HMAC authentication and a 16-byte `_encryption_key` for AES encryption. These keys are combined and base64-encoded to create the symmetric key. Despite the malware's attempt to conceal cryptographic material by nullifying the *crypter* object reference, the absence of forced garbage collection preserved it in memory besides other variables of the

RansomWare instance. Such variables include the RSA public key, filesystem paths (`sysRoot` and `localRoot`), and the victim's `publicIP` obtained from https://api.ipify.org. Note that we partially redacted our IP address to prevent information disclosure.

Fig. 11. Output of the *Py_Class* Plugin against *Python-Ransomware*

References

Ali-Gombe, A., Sudhakaran, S., Case, A., Richard III, G.G., 2019. {DroidScraper}: a tool for android {In-Memory} object recovery and reconstruction. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), pp. 547–559.

Ali-Gombe, A., Tambaoan, A., Gurfolino, A., Richard III, G.G., 2020. App-agnostic post-execution semantic analysis of android in-memory forensics artifacts. In: Proceedings of the 36th Annual Computer Security Applications Conference, pp. 28–41.

Case, A., 2011. Forensic memory analysis of android's dalvik vm. Source Seattle.

Case, A., Richard III, G.G., 2016. Detecting objective-c malware through memory forensics. Digit. Invest. 18, S3–S10.

Case, A., Richard III, G.G., 2017. Memory forensics: the path forward. Digit. Invest. 20, 23–33.

CISA, a, 2021. Ed 21-01: Mitigate SolarWinds Orion Code Compromise. https://www.cisa.gov/news-events/directives/ed-21-01-mitigate-solarwinds-orion-code-compromise, 2025-01-02.

CISA, b, 2021. Remediating Microsoft Exchange Vulnerabilities. https://www.cisa.gov/news-events/news/remediating-microsoft-exchange-vulnerabilities?utm_source=chatgpt.com, 2025-01-02.

CrowdStrike, 2024. Threat Actor Distributes Python-Based Information Stealer Using a Fake Falcon Sensor Update Lure. https://www.crowdstrike.com/en-us/blog/threat-actor-distributes-python-based-information-stealer/, 2025-01-05.

Crowe, J., 2021. Microsoft Exchange 0-day Vulnerabilities Mitigation Guide: what to Know & Do Now. https://www.ninjaone.com/blog/microsoft-exchange-0-day-vulnerabilities-mitigation/, 2025-01-02.

FBI, CISA, 2024. Known Indicators of Compromise Associated with Androxgh0st Malware. https://www.cisa.gov/news-events/alerts/2024/01/16/cisa-and-fbi-release-known-iocs-associated-androxgh0st-malware, 2025-01-15.

Foundation, P.S., a. CPython Internal Documentation: Frames. https://github.com/python/cpython/blob/main/InternalDocs/frames.md. Accessed: 2024-10-09.

Foundation, P.S., b. Garbage Collector Documentation. https://github.com/python/cpython/blob/main/InternalDocs/garbage_collector.md. Accessed: 2024-10-09.

Foundation, P.S., c. Python 3.8 Documentation. https://docs.python.org/3.8/. Accessed: 2024-10-10.

Foundation, P.S., d. What's New In Python 3.11. https://docs.python.org/3/whatsnew/3.11.html. Accessed: 2024-10-15.

Koutsokostas, V., Patsakis, C., 2021. Python and malware: developing stealth and evasive malware without obfuscation. arXiv preprint arXiv:2105.00565.

Li, N., Wang, S., Feng, M., Wang, K., Wang, M., Wang, H., 2023. Malwukong: towards fast, accurate, and multilingual detection of malicious code poisoning in oss supply chains. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 1993–2005.

Manna, M., Case, A., Ali-Gombe, A., Richard III, G.G., 2021. Modern macos userland runtime analysis. Forensic Sci. Int.: Digit. Invest. 38, 301221.

Manna, M., Case, A., Ali-Gombe, A., Richard III, G.G., 2022. Memory analysis of. net and. net core applications. Forensic Sci. Int.: Digit. Invest. 42, 301404.

Marcelo, Aira, Bren Matthew Ebriega, A.R., 2024. Python-based nodestealer version targets facebook ads manager. https://www.trendmicro.com/en_us/research/24/l/python-based-nodestealer.html, 2025-01-02.

Morag, A., 2022. Threat Alert: First python Ransomware Attack Targeting Jupyter Notebooks. https://www.aquasec.com/blog/python-ransomware-jupyter-notebook/, 2025-01-07.

ncorbuk, 2019. Python-Ransomware. https://github.com/ncorbuk/Python-Ransomware/tree/master, 2024-12-14.

Nelson, N., 2023b. Novel Pypi Malware Uses Compiled python Bytecode to Evade Detection. https://www.darkreading.com/application-security/novel-pypi-malware-compiled-python-bytecode-evade-detection, 2025-01-06.

Nelson, N., 2024. Citrine Sleet Poisons Pypi Packages with Mac and Linux Malware. https://www.darkreading.com/threat-intelligence/citrine-sleet-poisons-pypi-packages-mac-linux-malware, 2025-01-02.

News, T.H., 2023. Python-based Pyloose Fileless Attack Targets Cloud Workloads for Cryptocurrency Mining. https://thehackernews.com/2023/07/python-based-ploose-fileless-attack.html, 2025-01-05.

Nguyen, T.C., Vu, D.L., C. Debnath, N., 2024. An analysis of malicious behaviors of open-source packages using dynamic analysis. In: International Conference on Computer Applications in Industry and Engineering. Springer, pp. 102–114.

NightfallGT, 2021. BTC-clipper. https://github.com/NightfallGT/BTC-Clipper, 2024-12-14.

Oygenblik, D., Yagemann, C., Zhang, J., Mastali, A., Park, J., Saltaformaggio, B., 2024. {AI} psychiatry: forensic investigation of deep learning networks in memory images. In: 33rd USENIX Security Symposium (USENIX Security 24), pp. 1687–1704.

safesploit, 2023. Python Remote Administration Access (RAT). https://github.com/safesploit/PythonRAT/tree/main, 2024-012-14.

Schwartz, J., 2023. 'BlazeStealer' Python Malware Allows Complete Takeover of Developer Machines. https://www.darkreading.com/application-security/-blazestealer-python-malware-complete-takeover-developer, 2025-01-16.

Smmarwar, S.K., Gupta, G.P., Kumar, S., 2024. Android malware detection and identification frameworks by leveraging the machine and deep learning techniques: a comprehensive review. Telematics and Informatics Reports, 100130.

Sourcerer.io, . Python Internals: An Introduction. https://blog.sourcerer.io/python-internals-an-introduction-d14f9f70e583. Accessed: 2024-11-12.

Sylve, J., Case, A., Marziale, L., Richard, G.G., 2012. Acquisition and analysis of volatile memory from android devices. Digit. Invest. 8, 175–184.

Tam, K., Edwards, N., Cavallaro, L., 2015. Detecting android malware using memory image forensics. In: Engineering Secure Software and Systems (ESSoS) Doctoral Symposium.

tarcisio marinho, 2017. GonnaCry Ransomware. https://github.com/tarcisio-marinho/GonnaCry/tree/master, 2024-12-14.

The Volatility Foundation, 2017. The Volatility Framework: Volatile Memory Artifact Extraction Utility Framework. https://github.com/volatilityfoundation/volatility, 2024-12-01.

Volatility Foundation, . dwarf2json Repository. https://github.com/volatilityfoundation/dwarf2json. Accessed: 2025-01-2.

Wan, Y., 2024. PureHVNC Deployed via Python Multi-Stage Loade. https://www.fortinet.com/blog/threat-research/purehvnc-deployed-via-python-multi-stage-loader, 2025-01-16.

Wang, E., Zurowski, S., Duffy, O., Thomas, T., Baggili, I., 2022. Juicing v8: a primary account for the memory forensics of the v8 javascript engine. Forensic Sci. Int.: Digit. Invest. 42, 301400.

Williams, J., 2020. What You Need to Know about the SolarWinds Supply-Chain Attack. https://www.sans.org/blog/what-you-need-to-know-about-the-solarwinds-supply-chain-attack/, 2025-01-02.