Forensic Science International: Digital Investigation 49 (2024) 301757

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS USA 2024 - Selected Papers from the 24th Annual Digital Forensics Research Conference USA

On enhancing memory forensics with FAME: Framework for advanced monitoring and execution

Taha Gharaibeh [a,b,*], Ibrahim Baggili [a,b], Anas Mahmoud [b]

[a] Baggili(i) Truth (BiT) Lab, Center of Computation & Technology, Baton Rouge, LA, USA
[b] Division of Computer Science & Engineering, Louisiana State University, Baton Rouge, LA, USA

ARTICLE INFO

Keywords:
Memory forensics
Memory forensics tool testing
Interpreters (jitters)
Performance analysis
Tool speed testing framework

ABSTRACT

Memory Forensics (MF) is an essential aspect of digital investigations, but practitioners often face time-consuming challenges when using popular tools like the Volatility Framework (VF). VF, a widely-adopted Python-based memory forensics tool, presents difficulties for practitioners due to its slow performance. Thus, in this study, we evaluated methods to accelerate VF without modifying its code by testing four alternative Python Just In Time (JIT) interpreters - CPython, Pyston, PyPy, and Pyjion - using CPython as our baseline. Tests were conducted on 14 memory samples, totaling 173 GB, using a search-intensive VF plugin for Windows hosts. Employing our custom Framework for Advanced Monitoring and Execution (FAME), we deployed interpreters in Docker containers and monitored their real-time performance. A statistically significant difference was observed between the Python JIT interpreters and the standard interpreter. With PyPy emerging as the best interpreter, yielding a 15–20 % performance increase compared to the standard interpreter. Implementing PyPy has the potential to save significant time (many hours) when processing substantial memory samples. FAME enhances the efficiency of deploying and monitoring robust forensic tool testing, fostering reproducible research and yielding reliable results in both MF and the broader field of digital forensics.

1. Introduction

Memory Forensics (MF), a subdomain of Digital Forensics (DF), concentrates on acquiring (dumping or cloning) and analyzing volatile RAM (Case & Richard III 2017). This is particularly beneficial when investigating system intrusions involving stealthy malware that leaves no traces on a victim's hard drive. Random Access Memory (RAM) analysis reveals information such as active processes, stored usernames and passwords, open files, and active network connections. The widely adopted tool for RAM analysis is the open-source Volatility Framework (VF), version 3 (Ligh et al., 2014; Duke 2021; Balaoura 2018; Graziano et al., 2013). Other popular tools for extracting information from memory dumps include WinDbg, Rekall, and Varc. However, our study focuses on accelerating the VF.

One significant challenge in DF is the velocity, variety, and volume of digital evidence, resulting in substantial case backlogs for practitioners, sometimes up to three years (Scanlon 2016; van Baar et al., 2014; Baggili et al., 2014; McCullough et al., 2021; Sanchez et al., 2019). Solutions proposed include developing a robust DF talent pipeline and enhancing investigative technical solutions and processes. The exhaustive scans required for reconstructing memory artifacts necessitate faster VF processing for large workloads. VF is implemented in Python (Foundation 2023), which is typically slower than languages like C++ (Lion et al., 2022). One potential approach for improving runtime efficiency is migrating VF to a different programming language. However, this task is both complex and expensive, and the benefits may not outweigh the advantages of Python's extensibility and simplicity (Balreira et al., 2023; Tan et al., 2021). An alternative, cost-effective strategy is investigating whether VF's runtime can be optimized using different Python interpreters.

The standard Python implementation is CPython (interpreter + compiler). However, several interpreters and implementations have been developed throughout the years by the research community to overcome the shortcomings of the Python language. Our work studies four of these interpreters to determine if they have an effect on VF's performance. Specifically, we examined CPython, Pyston (Modzelewski 2023), PyPy (Team, 2019), and Pyjion (*Pyjion - A Python JIT Compiler*, 2022). To extensively test these interpreters, in a controlled and scalable setting, we had to devise a generalizable testing and monitoring framework that meets specific criteria outlined in our work.

* Corresponding author.
E-mail addresses: tahatlal@gmail.com (T. Gharaibeh), baggili@gmail.com (I. Baggili), amahmo4@lsu.edu (A. Mahmoud).

https://doi.org/10.1016/j.fsidi.2024.301757

Available online 5 July 2024

In DF, the ongoing challenge lies in robust testing forensic tools for scientific validity, error rates, and forensic soundness. This is vital not only for scientific integrity but also for the admissibility of digital evidence in the United States, following the Daubert Process and the Frye Test (Pan and Batten 2009; Horsman 2018, 2019; Mohamed et al., 2014; Baggili et al., 2007).

Thus, our work makes the following contributions.

- We propose a novel approach to enhance the performance of MF by accelerating VF in a cost-effective manner, without necessitating any code changes to the complex and widely-adopted software.
- We introduce Framework for Advanced Monitoring and Execution (FAME),[1] an open-source tool that facilitates real-time deployment and monitoring of Docker containers. FAME enables software deployment and monitoring experiments, enhancing continuous and robust testing of DF tools.
- In response to the lack of publicly available DF datasets (Grajeda et al., 2017), we contribute a public dataset consisting of over 173 GB of RAM samples and more than 750,000 data monitoring records obtained from our experiments[2,3].

In this paper, we explore accelerating VF by examining and substituting the Python Just-In-Time (JIT) interpreter with alternatives. We present background information on MF and VF, addressing our study's limitations. Our methodology involves demonstrating the problem, defining criteria, and guiding our research in identifying and constructing a solution. We then review related work, emphasizing the relevance, and importance of our research. We analyze the performance of Python JIT interpreters and highlight our approach's soundness. We discuss our findings, conclude our study, and suggest future research directions.

2. Background

This section includes background information about MF and how it relates to the VF, as well as the limitations and research questions.

2.1. Memory forensics

MF is a relatively young domain in the field of digital forensics. Vömel and Freiling (2012) explained that the main requirements for a forensically sound memory acquisition involve correctness, atomicity, and integrity. Atomicity is the concept that a single operation in memory, such as a read or write, is executed in one indivisible action without interruption. This ensures that the memory state maintains its correctness and integrity without corruption.

MF plays a crucial role in legal investigations, as it can yield evidence that is indispensable for substantiating or refuting a case. This type of analysis allows investigators to uncover valuable evidence that might be inaccessible through conventional forensic techniques, such as examining a device's hard drive. In many instances, a digital device's RAM can store crucial information unobtainable through other methods (e.g., searching the hard disk). For example, a suspect might have erased incriminating files from their hard drive, but remnants of those files could still linger in RAM. Moreover, the RAM can hold data about processes and activities occurring on the device during a crime (or at some point in time), such as the utilization of specific software or accessing particular websites.

After cloning a memory dump of RAM from the scene, it must be analyzed (Nyholm et al., 2022). Though the primary tool for RAM analysis is VF, alternative methods like string searches and pattern searches in memory using YARA signatures have also been employed (Cohen 2017). MF can be applied to any device containing RAM (Thing et al., 2010; Casey et al., 2019; Thomas et al., 2020, 2021; Wang et al., 2022; Manna et al., 2022). However, practitioners, such as DF investigators, must be mindful of the risks and legal aspects associated with handling digital evidence to ensure its admissibility in court.

Previous research has applied MF analysis to various devices, including mobile devices (Thing et al., 2010), Virtual Reality (VR) devices (Casey et al., 2019), cryptocurrency hardware wallets (Thomas et al., 2020), Universal Serial Bus (USB) attack platforms (Thomas et al., 2021), the V8 JavaScript (JS) engine (Wang et al., 2022), and. NET core applications (Manna et al., 2022). The efficiency and significance of MF are undeniable; however, the need for faster tools to address the growing DF backlog remains. Accelerating the tasks of incident responders engaged in investigative procedures is crucial, especially for those examining a machine following a discreet malware intrusion.

2.2. Volatility Framework

VF is an open-source MF framework initially developed by Aaron Walters, whose components are grounded in his academic research (Petroni et al., 2006; Walters and Petroni 2007). It is extensively utilized by various organizations, including the military, law enforcement, and incident response teams across all sectors (Case & Richard III 2017). VF is a robust MF tool that reconstructs information about a system's running processes and network connections while aiding in the identification of malicious processes. This extensible framework allows the use of built-in features and the integration of custom plugins. It enables human analysts to examine RAM dumps without requiring knowledge of low-level data reconstruction procedures. VF is regarded as the primary tool for system analysis. The framework can be used to extract information such as login credentials, IP addresses, and system configurations from the memory of a device (Wang et al., 2022; Lewis et al., 2018), recovering deleted files, identifying malware and rootkits, and reconstructing network activity (Sylve et al., 2012, Ligh et al., 2014).

2.3. Limitations & research questions

Our work was limited to a specific set of memory samples, which may not be fully representative of the complete spectrum of MF scenarios. Factors such as hardware specifications and Operating System (OS) may also impact Python interpreters' performance, aspects that may not have been thoroughly examined in our investigation. However, our scenarios were all deployed on the same hardware and with the same Docker image in a controlled environment. Our work did not take into account the effects of interpreter-specific optimizations and configuration alternatives, which may influence performance outcomes. We opted for the default settings across all deployments. Lastly, we investigated Python interpreters on a local cluster and batch containers and we aimed to answer the following research questions.

- **RQ1:** How does the replacement of Python JIT interpreters with the proposed framework impact the performance of VF in conducting MF?
- **RQ2:** How can the proposed FAME tool be integrated into reproducible research and experimental workflows to support memory forensics across various scenarios?

3. Methodology

In this section, we present the architecture of FAME and discuss how our objectives influenced the decision-making process. We demonstrate how a controlled testing environment can be employed to collect performance data for Python JIT interpreters. Moreover, we explain the process of memory sample curation. Furthermore, we describe the

[1] Code: https://lsu.box.com/s/w67xhdwa6mz9f2bbzof9sm2udbzilsql.
[2] Data samples link: https://bit.ly/3w60Fc4.
[3] Docker Excel sheet, Pypy3 & the volatility3: https://lsu.box.com/s/x54or67d0oa450gzwlon1mpom6cauu40.

comprehensive deploying and monitoring framework methodology. This section concludes by showcasing the framework in action, to analyze the data collected and evaluate the chosen Python JIT interpreters.

3.1. Problem generalization

We present the selection criteria for tested Python JIT interpreters and an overview of the methodology.

3.1.1. Python selection criteria

Previous research benchmarked Python JIT interpreters using synthetic data or simple queries, not reflecting real-world scenarios, especially for a full system (Roghult 2016; Juneau et al., 2010; Behnel et al., 2010). We tested Python JIT interpreters against the VF using realistic scenarios while considering the following criteria.

- **JIT Compiler Compatibility**: The tested software (VF) must run on Python and other Python JIT compilers. Compatibility implies no code changes, special tags, or metadata for guiding JIT compiler code compilation.
- **Zero Code Change**: No code changes, special tags, or metadata alterations needed for VF.
- **Extensible Testing Framework**: The framework should test time efficiency and forensic soundness, accommodating updates to JIT compilers and VF code.

We identified three Python JIT interpreters meeting our criteria: Pyston, PyPy, and Pyjion. These interpreters required minimal or no code changes. Note that initially, PyPy was incompatible (see Section 4.3).

3.1.2. Software framework specification

We aimed to develop a framework for scalable deployment and monitoring. Thus, we created FAME, designed based on the following criteria.

C1: **Commands Deployment**: Allows command deployment in isolated runtimes using containerization for controlled testing environments.
C2: **Lightweight**: Ensures efficient resource utilization, e.g., minimal Central Computing Unit (CPU) usage.
C3: **Monitoring Data Persistence**: Offers real-time container monitoring and access to historical performance data.
C4: **Easy to Customize**: Supports customization for extensibility, such as comparing output hash values for forensic soundness.

Our goal was a streamlined test deployment process and container scheduling for continuous, unattended operation, allowing researchers to focus on other tasks.

3.1.3. Methodology overview

The framework employed Docker containers as nodes. Each container executed a Python JIT interpreter with user-sourced commands, such as VF operations. Data was logged to measure runtime and other metrics. Timestamps were taken during tests, with the ability to measure completion time. Tests were conducted on a single machine, an MSI Katana GF66 11UE, with the following specifications:
Processor: MSI Intel ® Core ™ i7-11800H @ 2.30Ghz
Cores: 8.
L1 Cache: 640 KB.
L2 Cache: 10 MB.
L3 Cache: 24 MB.
Memory: 16 GB (2 × 8 GB) 3200Mhz DDR4.

3.2. FAME development

In this section, we discuss the development of the FAME framework. FAME was designed with the following objectives in mind.

- Ensuring consistent hardware and OS components to treat them as Independent Variable (IV)s. By maintaining a dockerized, stateless baseline without any resource manipulation, FAME produces more accurate results. In simpler terms, it enables a fair comparison between the different interpreters.
- Prioritizing portability and extensibility.
- Monitoring generalized performance metrics.

Our aim was to conduct tests in a noise-free, replicable environment. In the following sections, we elaborate on how these objectives were achieved.

3.2.1. FAME architecture

The architecture for monitoring Docker containers consists of three components: Main, Observer, and Publisher. Fig. 1 illustrates the layout of our monitoring and testing approach, showing the four tested Python JIT interpreters and the command execution within the container. FAME can be containerized or run on bare metal. The Main component, similar to the *main-node* architecture (see Fig. 2), serves as the coordinator and interface between the client and other components.

Listing 1. : JSON Data for Docker Container Configuration: Payload Sent in Each Deployment.

```
{
  "containerInfoDtos": [
    {
      "command": "python3 /app/volatility3/vol.py –f /app/
          memory_dumps/Windows10–Snapshot3.vmem
          windows.poolscanner.PoolScanner",
      "imageNameWithTag": "pytestimage:test"
    }
  ],
  "volumeModel": {
    "absolutePathOnHost": "$FOLDER_PATH",
    "mountPathInContainer": "/app"
  }
}
```

3.2.2. FAME deployment and monitoring

The Main component comprises two core libraries: (1) the open-source *docker-java* library, which builds the Docker client, and (2) *spring-boot-starter-webflux*, which enables the creation of non-blocking Application Programming Interface (API)s and reactive streams. User commands are processed and forwarded to the appropriate component. To run containers, users send an HTTP *POST* request to the Main component endpoint (see Fig. 2) specifying the payload (see Listing 1, *POST* endpoint payload). The Main component instructs Docker daemon (also known as Docker Engine API) to create the defined container (see Fig. 1) and informs the Observer component about the new container to monitor (see Fig. 2). The Main will schedule the containers in batches, with the ability to configure the time and the number of containers to run at once from the given commands that will be sent along in the HTTP *POST* payload (see Listing 1).

The Observer component (see Fig. 2) is a daemon actively collecting statistics and logs from the Docker daemon. It pushes the collected data to a non-relational database, MongoDB. The use of distributed messaging system (e.g., Kafka) was going to be implemented, however, the existing *spring-boot-starter-data-mongodb-reactive* library, enabled us to use and listen to changes (in collections) in real-time. These reactive streams simplified the architecture. The Publisher component, another

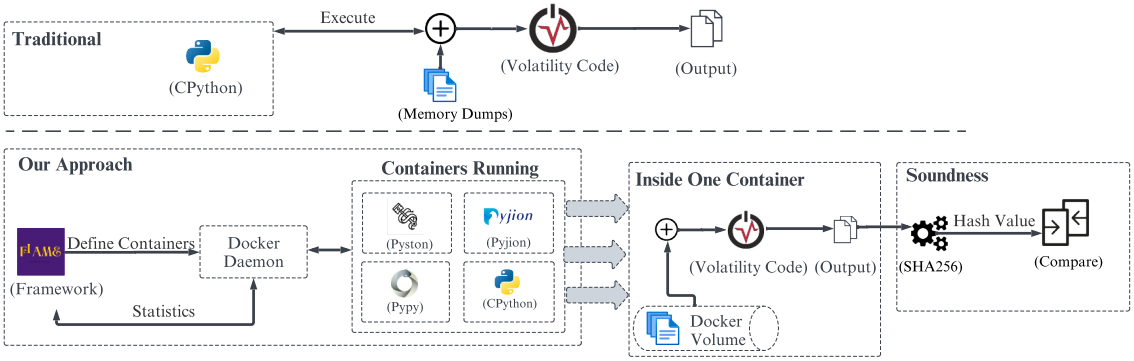

Fig. 1. High-level methodology: Comparing traditional vs. Our approach to deployment & execution of volatility code, containerized python JIT interpreters, and output result file hashing for soundness verification.

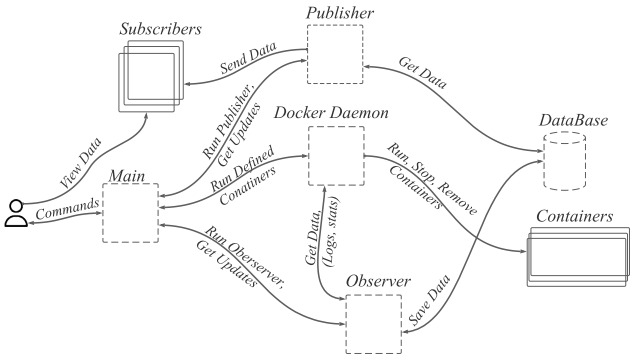

Fig. 2. FAME System Architecture: Deployment, Observation, and Publishing - User Commands Deploy Containers via Main Component & Performance Data is Accessed through Subscribers.

daemon, listens for changes in collections in real time and pushes data to active subscribers, as shown in Fig. 2. This systematic approach allows for efficient container monitoring at scale. Historical data can be retrieved or exported via the Publisher API, and readings are time-stamped and tagged with the container ID for clarity and time series chart visualization.

After building the deployment and monitoring backbone (see Fig. 2), a user is able to modify or add custom endpoints on both ends, the Publisher and Observer. FAME offers a lightweight protocol that does not require the opening of a new connection every time there is new data, when the connection is opened, all data will be sent on the same connection. It provides two data retrieval modes from the Publisher: sending duplicate data (active data retrieval) to be more efficient or sending data only when there are changes (e.g., the readings change) with that adding overhead of sending data and ensuring consistent connection from the Publisher.

3.2.3. Data output

As we mentioned earlier, data is retrieved via the Publisher after the subscriber's subscription. Additionally, subscribers may be required to develop a custom plugin that accommodates reactive streams. Data can be acquired in JSON format through MongoDB connectors and exported using the command line tool *mongoexport*, or alternatively, exported via the Publisher as JSON using previously stated modes. The statistics retrieved from a Publisher summary API are shown in the Listing 2.

Listing 2. : Java Class for Docker Statistics Summary Data.

```
public class DockerStatsSummaryDto {
    private String containerId;
    private float cpuPercent;
    private float memoryUsage;
    private float memoryLimit;
    private float networkI;
    private float networkO;
    private Date timestamp;
}
```

3.2.4. FAME validation

Apart from this, the data were directly acquired from the Docker daemon, utilizing the *docker-java* library. In order to verify the accuracy of FAME, code unit tests and visual inspection, a comparison test was conducted between the FAME statistics and the Docker stats, evaluating the congruence of the data. This comparison substantiated the correctness and reliability of the performance metrics and monitoring processes incorporated within FAME. Subsequently, multiple tests were carried out to confirm the consistency of the output and the framework's dependability. The design of FAME facilitates the replication of the test environment across a variety of systems and settings, thus enhancing the validity of the performance evaluation for different Python JIT compilers.

3.3. Tests

Before running performance tests on a system, it is critical to remove all user processes. User processes can interfere with the tests and produce inaccurate results, which can compromise the validity of the analysis. The test employed one scenario to be replicated on containers. The *System UTC clock* will be queried to obtain the current instant in Java, to timestamp the statistics and logs, accompanied by saving and publishing them in real time.

VF is a very rich framework and comprises plugins such as Pool Tag Scanning (PTS) that examine the memory dump for predefined patterns of bytes known as *pool tags*. The OS uses *pool tags* to manage memory allocation. The PTS process is time-consuming, as it involves an exhaustive search of the physical memory of a Windows system. This can take a significant amount of time to complete, depending on the size and complexity of the memory pool. It's a core plugin for other plugins (e.g., *netscan* that output network connections). In this paper, we use PTS, as the baseline test since it requires a lot of time to finish (a test on a single sample), as well as being a primitive building block. Typically, examiners will have to process multiple memory samples, so every minute counts.

Our tests will employ PTS, to test and find the median time needed to finish a run (end to end on all samples) (see Section 3.5), the CPU, and memory usages. However, to make use of the raw data acquired from the Docker daemon, we had to transform them (see Section 3.4). The median doesn't filter the data, rather, it is a robust representation of central tendency and less sensitive to outliers. The use of isolated runtimes, Docker containers, enables us to replicate the tests across all containers and to scale out the test on multiple containers at once while being efficient in resource usage (containers share the same OS resources). Finally, the tests can be replicated on other machines by just sharing the Docker image and having the Docker daemon running on the host.

The real-time reactive stream allowed us to examine how Docker containers behaved while the tests were running. This can assist researchers in identifying bottlenecks, making informed decisions, and identifying trends in order to enhance and optimize the tool being examined.

3.4. Metrics

In this section, we examine the raw data acquired from Docker daemon using Observer (see Fig. 2), the equations employed for data transformation and the presentation of the data. The *Statistics* class presents the raw data from Docker containers, implementing a serializable object containing multiple performance metrics like CPU usage, memory usage, network usage, and others. These metrics facilitate the analysis of Python JIT interpreters performance and their effect when used with the VF. Memory usage includes cache usage.[4] The *getStatsSummary* endpoint in Publisher is utilized to transform the raw data, taking the raw *Statistics* object along with supplementary information such as container ID, timestamp, name, and ID (see Listing 2). Several performance metrics are computed, including.

- **CPU Usage Percentage**: The difference between the current and previous total CPU usage is divided by the time interval and multiplied by 100 to derive the percentage of CPU usage.
- **Memory Usage (MB)**: Memory usage is divided by the constant *MB SIZE* to convert the value to megabytes.
- **Network Received Data (MB)**: Received bytes on the network are divided by the constant *MB SIZE* to compute the network received data in megabytes.
- **Network Transmitted Data (MB)**: The transmitted bytes on the network are divided by the constant *MB SIZE* to compute the network transmitted data in megabytes.

Metrics, formatted to two decimal places for clarity, are encapsulated in a *DockerStatsSummaryDto* object (see Listing 2), detailing a container's performance metrics, including container ID, timestamp, and name. In case of errors, an error is logged, and an empty *Optional* object is returned. The data, which can be visualized as tables or charts, provides insights into Python JIT interpreters performance with the VF, aiding in the selection of the most efficient interpreter for specific use cases.

3.5. Test data

In our experiment, we generated and acquired memory dumps for testing, using *VMware Workstation Pro* for data acquisition and curation. We first defined a Virtual Machine (VM) with a specific configuration, deployed on a Windows 10 operating system image. Memory samples of varying sizes were determined using Equation (1). Table 1 lists the memory dumps tested with Python JIT interpreters and the VF. Samples one to eleven were calculated by incrementing N, while the last three covered larger memory samples to assess interpreter performance with different sample sizes. The virtual machine had the following specifications.

Table 1
Acquired memory dump dataset.

Sample Name	Size [GB]	Operating System
S1	2	Windows 10
S2	3	Windows 10
S3	4	Windows 10
S4	5	Windows 10
S5	6	Windows 10
S6	7	Windows 10
S7	8	Windows 10
S8	9	Windows 10
S9	10	Windows 10
S10	11	Windows 10
S11	12	Windows 10
S12	16	Windows 10
S13	32	Windows 10
S14	48	Windows 10

- **Number of Virtual CPUs**: Two virtual CPUs were employed.
- **Amount of Allocated Memory RAM**: This is where we change the sample size, we used this Equation (1) to calculate the memory sample size.
- **Disk Type and Size**: NVMe hard disk type was used along with 60 GB memory size.
- **Network Settings**: Network Address Translation (NAT).

A custom Docker image was made to contain all the necessary dependencies to perform the tests, to assure constant system variables and dependencies across all containers. These dependencies include all the needed software (see Table A4) to perform the experiments, Python JIT interpreters and the needed modules for them to work.

$$\text{memorySampleSize} = N \times 2^{10} \tag{1}$$

Where: $N \in \mathbb{Z}$

4. Findings and evaluation

In this section, we present metrics and results of four Python JIT interpreters and their benchmarks. Task completion time depends on memory dump size, system complexity, and Python JIT interpreter operations. Larger memory dumps and complex systems may require more time using the VF. We validated each interpreter's output against the default interpreter using *SHA256* for forensic soundness. Statistical analysis was performed on data from multiple runs, totaling 60 h of processing.

4.1. Performance results

After we carried out the tests, more than 750, 000 thousand documents were collected and saved in the database, that's only the performance raw data. The logs, on the other hand, will be fetched all at once to generate the hashes, asynchronously on demand.

scipy, a Python library, was used to import the *stats* package to explore the distribution of data for normality. We tested the null hypothesis that a sample comes from a normal distribution. From the test, the α-value is typically at 0.05, which means that there is a 5 % chance of rejecting the null hypothesis when it is true. Our test showed that the data is not normally distributed; skewness and kurtosis were calculated for each Python JIT interpreter, along with their standard errors (SE) as depicted in Table A5.

These findings (shown in Table A5) shed light on the data distribution for each interpreter. The skewness and kurtosis ranges aid in

[4] https://github.com/docker-archive/libcontainer/pull/518#issue-32985796.

understanding the shape of the data distribution, which can be used for additional statistical analysis (e.g., determining the significance of the result). However, further study is needed to determine whether there is a significant difference between each group of data (e.g., CPU percentages), therefore we utilized Kruskal–Wallis test that is similar to ANOVA, data analysis, and non-parametric to evaluate and formulate the following hypothesis.

- **Data:** CPU usage for Python3, Pyston, Pyjion, PyPy3; not normally distributed.
- **Test:** Kruskal–Wallis.
- **Hypotheses:**
  - CPU Usage: H_0: Equal mean ranks.
  - Memory Usage: H_0: Equal mean ranks.
  - Completion Time: H_0: Equal mean ranks.

Table 2, indicates the statistics of each group on each Python JIT interpreter, is employed to depict the differences between each group. The significance of the data for each group can be observed. This comparison is made after determining that the data does not follow a normal distribution, which is established through the skewness and kurtosis in Table A5. Consequently, non-parametric statistical tests are utilized.

For each group, data was grouped by interpreter name and sample name, and then we calculated the median for CPU, memory usage and runtime. The grouped data object represents for each row within a unique combination of interpreter name and sample name, with median values computed over all rows belonging to that group. These values to be used and fed to the *ggbetweenstats* from the *ggstatsplot* package in R software. As a result, the median of medians will be retrieved, as depicted in Table 2.

A Kruskal–Wallis test (see Table 2) showed that the mean ranks of Python JIT interpreters are not the same. There is a significant difference between the CPU usage mean ranks **Chi square H(3) = 24.99, ρ< 0.001**, thereby rejecting the null hypothesis. Moreover, a significant difference is observed between the memory usage mean ranks **Chi square H(3) = 46.47, ρ< 0.001**, thereby rejecting the null hypothesis. Finally, a significant difference is found between the completion time mean ranks **Chi square H(3) = 10.72, ρ= 0.01**, thereby rejecting the null hypothesis. As a result, there is a significant difference in each group for the Python interpreters. The Kruskal–Wallis test was conducted to examine if there was a significant difference between the groups; however, it does not reveal the magnitude of the difference between subgroups (e.g., between Python and Pyston). To determine if a significant difference exists, Table 2 depicts the differences between each subgroup using the Dunn pairwise test, a post-hoc test and a Holm adjustment.

The results indicate a significant difference between PyPy3 and Python in CPU usage (since ρHolm-adj < 0.001). There is also a significant difference between Pyston and Python in CPU usage (since ρHolm-adj $<$ 0.001). Moreover, a significant difference is observed between PyPy3 and Python in the duration of the tests (since ρHolm-adj $= 0.01$). Similarly, a significant difference exists between PyPy3 and Python (see Table 2) in memory usage (since ρHolm-adj < 0.001), as well as between Pyjion and Python in memory usage (since ρHolm-adj < 0.001).

$$\text{difference} = \frac{\text{interpreter\_durations[i]} \quad - \quad \text{python3\_duration[i]}}{\text{python3\_duration[i]}} \times 100 \tag{2}$$

Where: $i. \in \mathbb{Z}$

python3_duration[i] = denotes the duration of the Python 3 baseline interpreter for the *i*-th run of the experiment, serving as a reference point for comparisons with the durations of other interpreters.
interpreter_durations[i] = signifies the duration of the *i*-th run of the experiment for a given interpreter, excluding Python 3. This value facilitates the assessment of the alternative interpreter's performance in relation to the baseline Python 3 interpreter.

Fig. 3 depicts the performance improvement achieved compared to the Python3 interpreter, with lower values indicating faster completion times. This comprehensive evaluation covers all samples (refer to Table 1). The "difference" variable, calculated using Equation (2), determines the percentage difference between the execution time of the baseline Python interpreter and other interpreters, enabling a performance comparison between them. By applying Equation (2) to every row in Fig. 3, the average performance improvements for PyPy, Pyston, and Pyjion are found to be 15.21 %, 10.93 %, and 7.43 %, respectively.

Performance variations may originate from both VF and system caching influences, as hinted in Fig. 4. This figure presents the test completion time rankings for all Python interpreters, suggesting the potential impact of caching behavior on performance. Similar observations have been reported in previous research (Roghult 2016). Although our study maintains constant hardware (described in Section 3.2), inconsistencies in performance are still noticeable, indicating unstable behavior. Nevertheless, the results ultimately converge, which may be attributed to the variability of processes with changing completion times. In summary, these findings underscore the importance of carefully selecting interpreters to achieve optimal performance in memory forensics.

4.2. Forensic soundness

Inherently, FAME ensures soundness by verifying the output integrity when using different Python JIT interpreters. A simple *string compare* method is insufficient to guarantee valid proof (i.e., **Hashes**) without properly running FAME. To formally establish the soundness property of FAME, we implement a procedure-based approach that ensures consistency in execution, yielding reliable and repeatable results. To achieve this, users must configure the containers and run FAME without interruptions. We define soundness results by hashing the outputs (e.g., process logs) from Section 3.2.3 using the *SHA256* function. The objective is for each Python JIT interpreter to execute the same procedure (see Fig. 1) and compare its results with those of the other interpreters. The Publisher component's hash string values, generated from the strings, match for all interpreters when running PTS. This indicates that the integrity of the data is preserved even after changing the Python JIT interpreter. The digests obtained from our experiments using the same sample align with those from other interpreters for that sample. Overall, these results confirm the forensic soundness of switching from one interpreter to another.

4.3. Encountered roadblocks

During our search for Python interpreters, we discovered that PyPy can not execute the most recent version of VF. According to the PyPy development team, the problem arises because Python performs less string compatibility tests if a C-written class does not include struct fields in comparison to its parent class (see Figure A5). As a result, it will not be considered as a solid class and will not work with PyPy since it does not support multiple class inheritance (PyPy is based on RPython).[5] We also encountered problems with Java string conversions and hashing libraries (e.g., Apache Commons Codec), causing incorrect hash values (completely wrong digest when compared to the standard implementation) in our unit tests. To resolve this, we implemented the SHA-256 algorithm using Java security library and processed raw bytes from Docker container logs as input, ensuring accurate results.

[5] https://foss.heptapod.net/PyPy/PyPy/-/issues/3821.

Table 2
Statistics of CPU, memory usage, and duration for python interpreters.

Measure	Dunn Test (Pairwise)				Kruskal–Wallis Test	*n*	Mean Rank	Median
	Python3	Pyston	PyPy3	Pyjion				
CPU (%) Median					χ^2 (3) = 24.99, $p < 0.001$, $\epsilon^2_{\text{ordinal}} = 0.45$	56		
Python3	–	9.31e-4*	1.96e-4*	0.65		14	42.18	88.76
Pyston	9.31e-4*	–	0.68	0.02*		14	19.14	86.88
PyPy3	1.96e-4*	0.68	–	6.12e-3*		14	16.57	86.34
Pyjion	0.65	0.02*	6.12e-3*	–		14	36.11	88.44
Memory Usage (MB) Median					χ2 (3) = 46.47, $p < 0.001$, $\epsilon^2_{\text{ordinal}} = 0.84$	56		
Python3	–	0.82	4.13e-8*	1.71e-3*		14	13.79	106.03
Pyston	0.82	–	1.33e-7*	3.00e-3*		14	15.21	106.22
PyPy3	4.13e-8*	1.33e-7*	–	0.05*		14	49.50	274.86
Pyjion	1.71e-3*	3.00e-3*	0.05*	–		14	35.50	149.33
Duration (Seconds) Median					χ^2 (3) = 10.72, $p = .01$, $\epsilon^2_{\text{ordinal}} = 0.19$	56		
Python3	–	0.08	0.01*	0.50		14	39.07	1547.10
Pyston	0.08	–	0.59	0.59		14	24.14	1304.38
PyPy3	0.01*	0.59	–	0.37		14	20.21	1306.16
Pyjion	0.50	0.59	0.37	–		14	30.57	1453.07

Where: *p < 0.05.

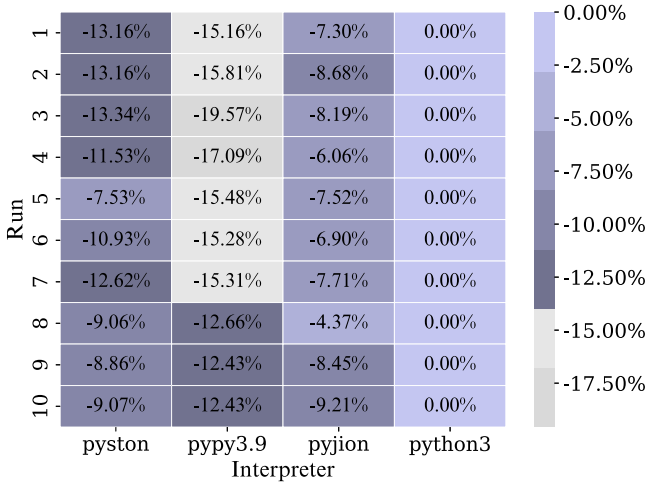

Fig. 3. Percentage Increase For Each Run Compared to the baseline Interpreter, Python3, Lower Values Indicate Better Performance.

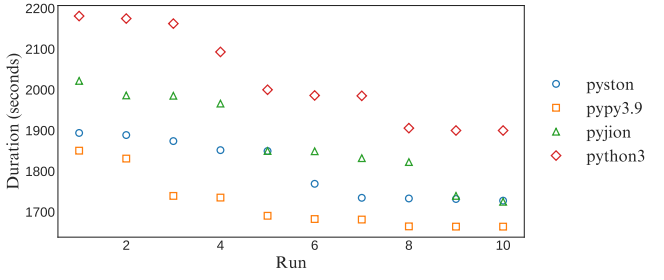

Fig. 4. The average time needed for a python JIT Interpreter to complete processing sample.

5. Related work

In this section, we show the related work on speeding up memory forensics, benchmarking the Python JIT interpreters, and tools that were developed to monitor Docker containers.

5.1. Speeding up memory analysis

Little work has been conducted on speeding up memory analysis. Memory analysis is a trade-off between time and accuracy. To improve accuracy and reduce time needed to analyze, the use of Intezer Analyze CLI was proposed (Holtzman 2020). It uses VF modules, such as *procdump* and *malfind*, to dump a process's executable and aid in the detection of possible memory injections and other code items that may be used to assess if a file is malicious or trusted. To speed up kernel memory allocations (Sylve et al., 2016), proposes a novel approach by scanning the memory pages that are associated with pool allocations, while maintaining a high level of accuracy.

Developing and testing new VF plugins can be time-consuming. Therefore, it's crucial to conduct tests for each new plugin to ensure accuracy. Researchers have found that creating a virtual volatile memory disk to hold memory dumps during tests and outputs can boost performance, claiming up to four times the speed of a regular hard drive when running VF plugins, though performance may vary depending on the plugin used (Tomchop 2014). However, the use of Ramdisks (Kind 2011) carries the risk of data loss due to their volatile nature.

5.2. Python JIT interpreters evaluation

To address the Python language execution speed, during the last decade, we have seen many Python JIT interpreter implementations. At the time of writing, there were more than twelve JIT Python compilers with different specifications. Some require code changes (Stefan Behnel, n.d.), and others perform better on specific problems (Roghult 2016), such as being more efficient in sorting a list. Past work developed a test's suite to measure the performance of four Python JIT interpreters and found that PyPy and Jython (Juneau et al., 2010) were the fastest for the majority of tests when running code using only Python syntax and data types (Roghult 2016). Then comes Cython (Behnel et al., 2010), which may require code modifications depending on the mode used. A scientifically reproducible method for testing different Python interpreters is to explore their runtime efficiency by running them in Docker containers and monitoring their behavior.

5.3. Docker monitoring tools

Monitoring assists DevOps teams in detecting and resolving issues, such as identifying the main causes of poor application performance by utilizing various crucial system resource metrics (e.g. CPU usage) and exploring Docker container artifacts (Boettiger 2015; Henkel et al., 2020; Hussain et al., 2017). Monitoring tools are vital not just for observability, but for also identifying resource bottlenecks and health system issues in an application (Cai and Kazman 2016). There are several trusted performance profilers in the research community (Casalicchio and Perciballi 2017) and tools available, including *docker*

Table 3
Docker monitoring tools vs. FAME criteria.

Tool	C1	C2	C3	C4
docker stats	×	✔	Limited	×
cAdvisor	×	✔	Real-time only	Difficult
Prometheus	×	Moderate	✔	Difficult
Netdata	×	✔	Real-time only	Limited
Datadog agent	×	Moderate	✔	Limited
ELK	×	×	✔	Moderate
FAME	✔	✔	✔	✔

Legend: C1 = Command Deployment, C2 = Lightweight, C3 = Monitoring Data Persistence, C4: Easily Customized.

stats, cAdvisor, Prometheus, Netdata, Datadog agent, ELK, and many others.

We describe and list past container monitoring solutions and compare them to FAME in Table 3: *(i)* **docker stats:** a command-line tool within Docker that provides a live stream of container resource usage statistics *(ii)* **cAdvisor (Container Advisor):** is an open-source solution that exposes container data through an agent actively collecting performance data from a Docker daemon *(iii)* **Prometheus:** a widely used open-source monitoring and alerting system, features a dimensional data model, flexible query language, an efficient time series database, and modern alerting approach *(iv)* **Datadog Agent:** another open-source agent-based monitoring tool for Docker containers that is installed as a node within the host where the Docker platform is operating. It is a Software as a Service (SaaS) that was developed by Datadog Inc to collect data from cloud-scale applications *(v)* **Netdata:** an open-source monitoring tool designed to collect real-time data, such as CPU usage and other useful metrics and *(vi)* **The ELK stack (Elasticsearch, Logstash, and Kibana):** an open-source tool focusing on logs, providing analytics and search functionalities via HTTP endpoints. Logstash pulls logs from Docker containers and applies custom filters, while Elasticsearch serves as the core search engine and Kibana offers visualization.

5.4. Related work summary & discussion

The literature lacks rigorous research on Python interpreters. To address this, we propose key steps for a comprehensive assessment. Firstly, testing on real-world systems ensures practicality. Secondly, employing extensive datasets covers diverse scenarios. Thirdly, using a reproducible research framework like FAME ensures consistency. Lastly, evaluating forensic soundness confirms result consistency across interpreters. This approach will enhance the robustness and relevance of Python interpreter evaluation research.

For our research to materialize, we had to abide by the FAME criteria presented in Table 3, and discussed in Section 3.1.2. With respect to the aforementioned monitoring tools (Section 5.3), FAME overcomes their shortcomings. It offers several distinct advantages over existing solutions by providing a lightweight, easy-to-use, and tailor-made solution. Moreover, the unique capability of testing the forensic soundness of logs from different Docker containers sets it apart from existing solutions, thereby making it suitable for reproducible research purposes (e.g. testing various Python JIT interpreters and their soundness). In summary, Table 3 shows our proposed FAME monitoring solution that addresses all four criteria (C1, C2, C3 and C4), whereas other tools either partially meet the criteria or do not meet them at all. Overall, our approach fills major gaps in the literature and monitoring tools that is both useful to the DF and forensic tools testing.

6. Discussion

Prior research by Casalicchio and Perciballi (2017) has investigated the overhead introduced by Docker containers concerning CPU usage in comparison to native execution. The findings suggested that the impact of containerization on CPU performance may be more significant at moderate utilization levels and decrease as computational resource demand increases. Notably, when CPU utilization lies between 65 % and 75 %, there is approximately a 10 % overhead compared to native CPU load. Interestingly, as CPU utilization exceeds 80 %, the overhead falls below 5 % (Casalicchio and Perciballi 2017).

All tests in our study were conducted on Docker containers, ensuring consistency in our results. We accounted for a 5 % performance margin when any interpreter's CPU usage exceeds 80 %, and a 10 % margin when CPU utilization lies between 65 % and 75 %. As seen in Table 3, Python3 exhibited the highest CPU utilization median, with a tendency to be higher, followed by Pyjion, Pyston, and PyPy. Conversely, as anticipated, PyPy demonstrated the worst memory usage among the Python JIT interpreters (see Table 3). PyPy is known for its higher baseline memory consumption compared to Python, and this value tends to increase as the JIT generates more machine code over time. However, it is expected to converge, indicating that memory usage should grow during program execution but only up to a certain maximum.

Our results are consistent with previous literature, which shows that PyPy outperforms other interpreters in terms of execution speed (Crapé and Eeckhout 2020; Roghult 2016). To illustrate the practical impact of these findings, consider a typical organization that needs to process a 1 TB memory sample using the VF. Based on the observed performance improvements with our apparatus, the organization can significantly reduce processing time by using PyPy instead of the standard Python interpreter.

For instance, in a typical organization with 1 TB of memory samples, if the standard Python interpreter takes 36 min per run for a 173 GB memory sample, PyPy could reduce the processing time to around 30 min per sample, saving 6 min per run. With 1 TB memory samples divided into roughly 5.78 collections of 178 GB samples ($\frac{1000}{178} \approx 5.78$), the total time savings would be approximately 34 min or 18.4 %. We note that this is a conservative estimate, and that in certain instances, the improvement can reach up to 40 %. This has the potential to save several hours of work when handling cases that involve a significant number of memory samples.

FAME significantly improves the efficiency of deploying and monitoring forensic tool testing on a large scale, ensuring experiment integrity through its forensic soundness. This study shows FAME's role in facilitating reproducible research within DF, emphasizing its potential to influence real-world memory analysis, especially when using tools like the VF in organizational setups.

7. Conclusions & future work

In this paper, we introduced a tool designed for the research community to explore and monitor containerized tools. We applied our tool to the use case of "towards faster memory forensics" and found that PyPy is the best-performing interpreter, being 15.2 % faster in completing a full run and exhibiting better CPU usage than standard Python. However, it comes with a higher memory usage cost, exceeding Python by 100 megabytes in our case (see **RQ1**). Our work streamlines the process of configuring, deploying, and monitoring Docker containers for researchers conducting stress tests or assessing tool performance (see **RQ2**).

In this paper, we investigated the acceleration of VF by replacing Python JIT interpreters. Our future work will explore building VF core plugins using C++ and injecting them as modules (extending Python with C or C++).[6] Additionally, we plan to integrate a messaging system into the FAME architecture to enhance scalability and improve the signaling protocols between the Main, Observer, and Publisher components.

[6] https://docs.python.org/3/extending/extending.html.

FAME empowers researchers to replicate and share their work within the community using Docker images for experiments. This approach promotes exploration and builds upon artifacts while ensuring consistent setup across machines. Our work benefits both the DF and forensic tool testing by promoting accessibility and reproducibility.

Appendix A. Tables & Figures

Table A.4
Software Utilized in the Experiments.

Software	Version
Operating System (MSI)	Windows 10
Operating System (Docker Container)	Ubuntu 20.04.5
CPython	3.7.5
Pyston	2.3.5
PyPy	3.9.0
Pyjion	2.0.0
Docker	4.3.0

Table A.5
Skewness and Kurtosis Ranges for Duration, CPU Percent, and Memory Usage.

Interpreter	Metric	Range
PyPy3	Duration	Skewness: 0.01–0.81 (SE = 0.20), Kurtosis: −0.35 – 1.25 (SE = 0.41)
	CPU Percent	Skewness: −0.95 to −0.93 (SE = 0.01), Kurtosis: −0.95 to −0.90 (SE = 0.01)
	Memory Usage	Skewness: 5.22–5.24 (SE = 0.01), Kurtosis: 78.61–78.65 (SE = 0.01)
Python3	Duration	Skewness: 0.33–1.13 (SE = 0.20), Kurtosis: 0.10–1.70 (SE = 0.41)
	CPU Percent	Skewness: −1.29 to −1.27 (SE = 0.01), Kurtosis: −0.17 to −0.13 (SE = 0.01)
	Memory Usage	Skewness: 3.46–3.48 (SE = 0.01), Kurtosis: 23.00–23.04 (SE = 0.01)
Pyjion	Duration	Skewness: 0.28–1.08 (SE = 0.20), Kurtosis: −0.17 – 1.42 (SE = 0.41)
	CPU Percent	Skewness: −1.26 to −1.24 (SE = 0.01), Kurtosis: −0.26 to −0.22 (SE = 0.01)
	Memory Usage	Skewness: 5.26–5.28 (SE = 0.01), Kurtosis: 66.29–66.34 (SE = 0.01)
Pyston	Duration	Skewness: 0.36–1.16 (SE = 0.20), Kurtosis: −0.85 – 0.75 (SE = 0.41)
	CPU Percent	Skewness: −1.02 to −1.00 (SE = 0.01), Kurtosis: −0.85 to −0.80 (SE = 0.01)
	Memory Usage	Skewness: 4.21–4.24 (SE = 0.01), Kurtosis: 41.38–41.42 (SE = 0.01)

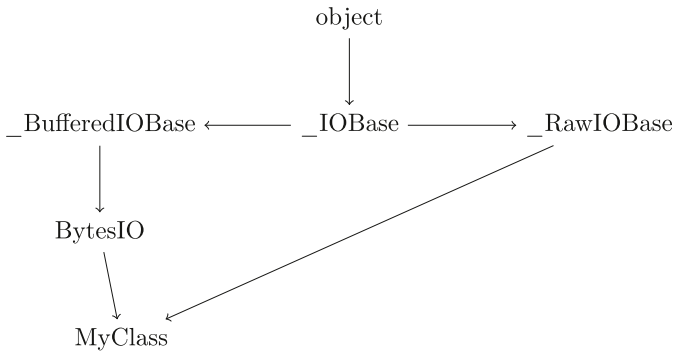

Figure A.5. Inheritance Conflicts in Instance Layout.

References

Baggili, I.M., Mislan, R., Rogers, M., 2007. Mobile phone forensics tool testing: a database driven approach. International Journal of Digital Evidence 6 (2), 168–178.
Baggili, I., Marrington, A., Jafar, Y., 2014. Performance of a logical, five-phase, multithreaded, bootable triage tool. In: Advances in Digital Forensics X: 10th IFIP WG 11.9 International Conference, Vienna, Austria, January 8-10, 2014, Revised Selected Papers 10'. Springer, pp. 279–295.
Balaoura, S., 2018. Process Injection Techniques and Detection Using the Volatility Framework. University of Piraeus, Greece. PhD thesis.
Balreira, D.G., Silveira, T.L.d., Wickboldt, J.A., 2023. Investigating the impact of adopting python and c languages for introductory engineering programming courses. Comput. Appl. Eng. Educ. 31 (1), 47–62.
Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D.S., Smith, K., 2010. Cython: the best of both worlds. Comput. Sci. Eng. 13 (2), 31–39.
Boettiger, C., 2015. An introduction to docker for reproducible research. SIGOPS Oper. Syst. Rev. 49 (1), 71–79.
Cai, Y., Kazman, R., 2016. Software architecture health monitor. In: Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities', BRIDGE, vol. 16, pp. 18–21.
Casalicchio, E., Perciballi, V., 2017. Measuring docker performance: what a mess. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion', pp. 11–16.
Case, A., Richard III, G.G., 2017. Memory forensics: the path forward. Digit. Invest. 20, 23–33.
Casey, P., Lindsay-Decusati, R., Baggili, I., Breitinger, F., 2019. Inception: virtual space in memory space in real space–memory forensics of immersive virtual reality with the htc vive. Digit. Invest. 29, S13–S21.
Cohen, M., 2017. Scanning memory with yara. Digit. Invest. 20, 34–43.
Crapé, A., Eeckhout, L., 2020. A rigorous benchmarking and performance analysis methodology for python workloads. In: 2020 IEEE International Symposium on Workload Characterization (IISWC)', pp. 83–93.
Duke, J.E., 2021. Memory Forensics Comparison of Apple M1 and Intel Architecture Using Volatility Framework.

Foundation, P.S., 2023. Welcome to Python.org. https://www.python.org. (Accessed 1 February 2023).
Grajeda, C., Breitinger, F., Baggili, I., 2017. Availability of datasets for digital forensics–and what is missing. Digit. Invest. 22, S94–S105.
Graziano, M., Lanzi, A., Balzarotti, D., 2013. Hypervisor memory forensics. In: International Symposium on Recent Advances in Intrusion Detection'.
Henkel, J., Bird, C., Lahiri, S.K., Reps, T., 2020. Learning from, understanding, and supporting devops artifacts for docker. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering'. ICSE '20, pp. 38–49.
Holtzman, S., 2020. Accelerate memory forensics with intezer analyze - intezer. https://www.intezer.com/blog/malware-analysis/accelerate-memory-forensics/. (Accessed 25 January 2023).
Horsman, G., 2018. "'i couldn't find it your honour, it mustn't be there!"–tool errors, tool limitations and user error in digital forensics'. Sci. Justice 58 (6), 433–440.
Horsman, G., 2019. Tool testing and reliability issues in the field of digital forensics. Digit. Invest. 28, 163–175.
Hussain, W., Clear, T., MacDonell, S., 2017. Emerging trends for global devops: a New Zealand perspective. In: Proceedings of the 12th International Conference on Global Software Engineering. ICGSE '17, pp. 21–30.
Juneau, J., Baker, J., Wierzbicki, F., Muoz, L.S., Ng, V., Ng, A., Baker, D.L., 2010. The Definitive Guide to Jython: Python for the Java Platform.
Kind, T., 2011. Ramdisk Benchmarks', vol. 52. University of California.
Lewis, N., Case, A., Ali-Gombe, A., Richard III, G.G., 2018. Memory forensics and the windows subsystem for linux. Digit. Invest. 26, S3–S11.
Ligh, M., Case, A., Levy, J., Walters, A., 2014. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory.
Lion, D., Chiu, A., Stumm, M., Yuan, D., 2022. Investigating managed language runtime performance: why {JavaScript} and python are 8x and 29x slower than c++, yet java and go can be faster?. In: 2022 USENIX Annual Technical Conference (USENIX ATC 22), pp. 835–852.
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.
McCullough, S., Abudu, S., Onwubuariri, E., Baggili, I., 2021. Another brick in the wall: an exploratory analysis of digital forensics programs in the United States. Forensic Sci. Int.: Digit. Invest. 37, 301187.
Modzelewski, K., 2023. The Pyston blog. https://blog.pyston.org. (Accessed 1 February 2023).
Mohamed, A.F.A.L., Marrington, A., Iqbal, F., Baggili, I., 2014. Testing the forensic soundness of forensic examination environments on bootable media. Digit. Invest. 11, S22–S29.
Nyholm, H., Monteith, K., Lyles, S., Gallegos, M., DeSantis, M., Donaldson, J., Taylor, C., 2022. The evolution of volatile memory forensics. Journal of Cybersecurity and Privacy 2 (3), 556–572.
Pan, L., Batten, L.M., 2009. Robust performance testing for digital forensic tools. Digit. Invest. 6 (1–2), 71–81.
Petroni, N.L., Walters, A., Fraser, T., Arbaugh, W.A., 2006. Fatkit: a framework for the extraction and analysis of digital forensic data from volatile system memory. Digit. Invest. 3 (4), 197–210.
Roghult, A., 2016. Benchmarking python Interpreters : Measuring Performance of cpython, Cython, Jython and Pypy.
Sanchez, L., Grajeda, C., Baggili, I., Hall, C., 2019. A practitioner survey exploring the value of forensic tools, ai, filtering, & safer presentation for investigating child sexual abuse material (csam). Digit. Invest. 29, S124–S142.
Scanlon, M., 2016. Battling the digital forensic backlog through data deduplication. In: 2016 Sixth International Conference on Innovative Computing Technology (INTECH)', IEEE, pp. 10–14.
Stefan Behnel, R.B.. Cython: C-extensions for python. n.d. https://cython.org/. (Accessed 25 January 2023).
Sylve, J., Case, A., Marziale, L., Richard, G.G., 2012. Acquisition and analysis of volatile memory from android devices. Digit. Invest. 8, 175–184.
Sylve, J.T., Marziale, V., Richard III, G.G., 2016. Pool tag quick scanning for windows memory analysis. Digit. Invest. 16, S25–S32.
Tan, J., Chen, Y., Liu, Z., Ren, B., Song, S.L., Shen, X., Liu, X., 2021. Toward efficient interactions between python and native libraries. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering', pp. 1117–1128.
Team, T., 2019. PyPy. PyPy. https://www.pypy.org.
Thing, V.L., Ng, K.-Y., Chang, E.-C., 2010. Live memory forensics of mobile phones. Digit. Invest. 7, S74–S82.
Thomas, T., Piscitelli, M., Nahar, B.A., Baggili, I., 2021. Duck hunt: memory forensics of usb attack platforms. Forensic Sci. Int.: Digit. Invest. 37, 301190.
Thomas, T., Piscitelli, M., Shavrov, I., Baggili, I., 2020. Memory foreshadow: memory forensics of hardware cryptocurrency wallets–a tool and visualization framework. Forensic Sci. Int.: Digit. Invest. 33, 301002.
Tomchop, 2014. Speeding up volatility with ramdisks · tomchop. http://tomchop.me/2014/09/01/speeding-up-volatility-ramdisks/. (Accessed 26 January 2023).
van Baar, R.B., van Beek, H.M., Van Eijk, E., 2014. Digital forensics as a service: a game changer. Digit. Invest. 11, S54–S62.
Vömel, S., Freiling, F.C., 2012. Correctness, atomicity, and integrity: defining criteria for forensically-sound memory acquisition. Digit. Invest. 9 (2), 125–137.
Walters, A., Petroni, N.L., 2007. Volatools : Integrating Volatile Memory Forensics into the Digital Investigation Process.
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.