FreePart: Hardening Data Processing Software via Framework-based Partitioning and Isolation

Ali Ahad
University of Maryland
College Park, MD, USA
aahad@umd.edu

Gang Wang
University of Illinois at Urbana-Champaign
Urbana, IL, USA
gangw@illinois.edu

Chung Hwan Kim
University of Texas at Dallas
Richardson, TX, USA
chungkim@utdallas.edu

Suman Jana
Columbia University
New York, NY, USA
suman@cs.columbia.edu

Zhiqiang Lin
Ohio State University
Columbus, OH, USA
zlin@cse.ohio-state.edu

Yonghwi Kwon
University of Maryland
College Park, MD, USA
yongkwon@umd.edu

ABSTRACT

Data processing oriented software, especially machine learning applications, are heavily dependent on standard frameworks/libraries such as TensorFlow and OpenCV. As those frameworks have gained significant popularity, the exploitation of vulnerabilities in the frameworks has become a critical security concern. While software isolation can minimize the impact of exploitation, existing approaches suffer from difficulty analyzing complex program dependencies or excessive overhead, making them ineffective in practice.

We propose FreePart, a framework-focused software partitioning technique specialized for data processing applications. It is based on an observation that the execution of a data processing application, including data flows and usage of critical data, is closely related to the invocations of framework APIs. Hence, we conduct a temporal partitioning of the host application's execution based on the invocations of framework APIs and the data objects used by the APIs. By focusing on data accesses at runtime instead of static program code, it provides effective and practical isolation from the perspective of data. Our evaluation on 23 applications using popular frameworks (*e.g.*, OpenCV, Caffe, PyTorch, and TensorFlow) shows that FreePart is effective against all attacks composed of 18 real-world vulnerabilities with a low overhead (3.68%).

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Software Isolation; Software Partitioning; Data Processing Frameworks

ACM Reference Format:
Ali Ahad, Gang Wang, Chung Hwan Kim, Suman Jana, Zhiqiang Lin, and Yonghwi Kwon. 2023. FreePart: Hardening Data Processing Software via Framework-based Partitioning and Isolation. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3623278.3624760

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 979-8-4007-0394-2/23/03.
https://doi.org/10.1145/3623278.3624760

1 INTRODUCTION

Data processing software, particularly machine learning and image processing software, are widely used to support critical systems in practice. These programs are heavily dependent on data processing frameworks and libraries. For example, OpenCV [68], an open-source computer vision toolkit, has over 18 million downloads and is widely used by real-world systems [37]. Machine learning based applications also rely on a few well-known frameworks such as PyTorch [76], Caffe [27], TensorFlow [89], and Scikit-learn [83]. The wide adoption of these frameworks leads to an unfortunate consequence that vulnerabilities in them could significantly impact the host applications [23, 49, 85, 86, 99]. In particular, software vulnerability is one of the critical attack surfaces as it can affect the entire host application's memory (*i.e.*, code and data). Compared to the vulnerabilities in ML models/data that affect the decision made by the framework, software vulnerabilities can *allow attackers to do almost anything*. For instance, software vulnerabilities in OpenCV are often considered to have a *high security severity* as they can affect various critical systems using OpenCV [4], regardless of algorithms and models used by the host application.

Software Isolation and Limitations. Software isolation [13, 50, 51, 106] can mitigate the exploitation of vulnerabilities by partitioning software into multiple parts and executing each part in a separate and isolated process or using intra-process isolation technique [41, 97]. With the technique, the impact of the exploitation of a vulnerability is confined to an isolated partition so that the rest of the program is protected.

However, those approaches suffer from limited security for critical data or APIs, runtime performance overhead, and/or require accurate dependence analysis (otherwise breaking the program's functionalities). This is because, in part, program code and data that need to be partitioned are intertwined.

Observations. We observe that the *execution and data accesses* of data processing software are *tightly correlated to the framework APIs* invoked. In other words, the invocation of framework APIs

at runtime depicts the program execution's *temporal progress*, implying *what data to be read or written* at a certain point of the execution. For example, a program first executes *data loading APIs* to load the input data, followed by a series of data pre-processing tasks. It then calls *data processing APIs* (*e.g.*, ML algorithms), followed by the data interpretation/post-processing logic. Finally, *data visualization/storing APIs* are called to present or save the results.

Framework-based Execution Partitioning. Based on the observations, we propose a program partitioning/isolation technique *from the perspective of framework APIs' execution and correlated data*. Specifically, we categorize the APIs into four types reflecting their high-level purposes, following a typical workflow of a data-processing application: *loading*, *processing*, *visualizing*, and *storing* APIs. Then, we execute each type of framework API on a separate and isolated process. We track which type of API is executed to infer the permission of the critical data. Lastly, we restrict the privileges of each process by only allowing necessary system calls for framework APIs.

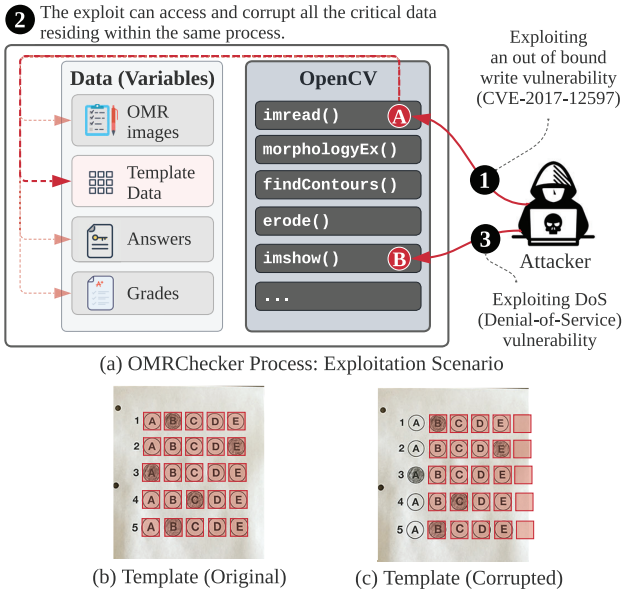
Challenges. We solve three major challenges. First, we conduct a systematic analysis to support our design of framework-based partitioning and isolation (*e.g.*, four types of framework APIs in Section 4.1). Second, we develop dynamic and static analysis techniques to automatically categorize hundreds of framework APIs based on their data dependency patterns. task (Section 4.2). Third, we reduce the runtime performance overhead caused by inter-process communications by enabling direct data sharing between the partitioned processes via the lazy data copy technique (Section 4.3.2).

Our major contributions are summarized as follows:

- We propose a *framework-based execution partitioning and isolation approach* and carefully design and implement a proof-of-concept system, FreePart.
- We develop (1) a hybrid profiling technique to automatically categorize framework APIs and (2) a lazy data copy technique to reduce the performance overhead.
- We apply FreePart to four widely used data processing frameworks (OpenCV, Caffe, PyTorch, and TensorFlow), that demonstrate the generality of our approach.
- We evaluate FreePart's performance on 23 applications and attacks composed of 18 real-world vulnerabilities from Common Vulnerabilities and Exposures (CVEs). FreePart effectively prevents all the attacks with a low runtime overhead (3.68%).

2 THREAT MODEL

We assume an attacker who exploits a software vulnerability in a target data processing framework, such as TensorFlow or OpenCV. The attacker invokes a framework API with a maliciously crafted input to exploit a vulnerability, such as a memory corruption vulnerability. Note that while such a vulnerability exists within a framework API, an attacker can exploit it to disrupt the entire host application due to the lack of isolation between the framework and application. For example, by exploiting it the attacker may execute malicious code in the host application process, to corrupt critical data, or crash the host application process for a denial-of-service attack. We do not assume attackers can compromise the underlying system software such as the operating system, as we rely on the isolation of processes enforced by the OS kernel. We also trust our runtime support as it is protected via the OS kernel.



(a) OMRChecker Process: Exploitation Scenario

(b) Template (Original)

(c) Template (Corrupted)

Figure 1: Attack Scenario of the Motivating Example.

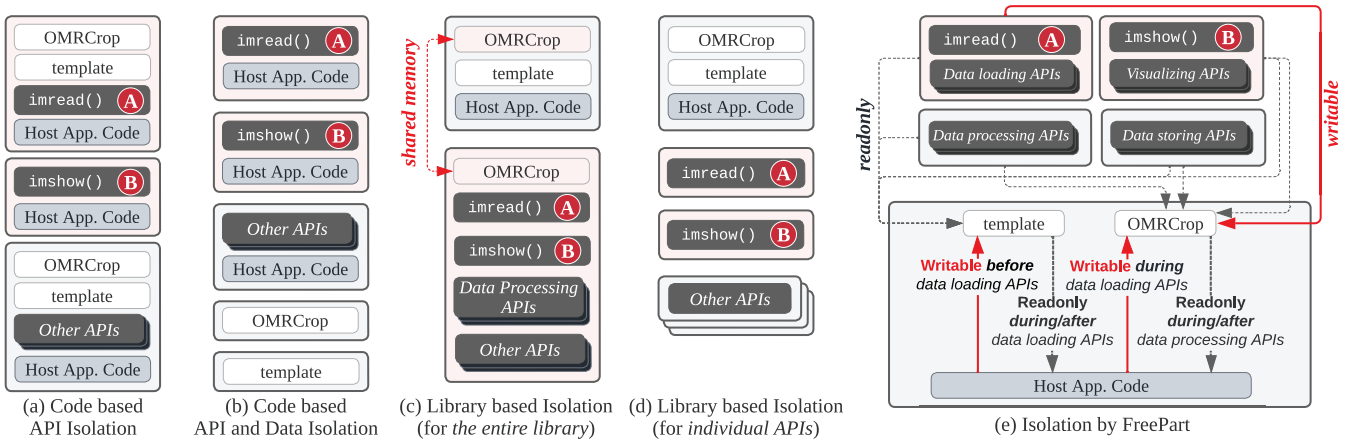
Scope. Attacks that exploit a host application's vulnerability (not the framework's vulnerability) are outside of our scope. Attacks targeting machine learning (ML) algorithms (*e.g.*, adversarial example attacks [6, 9, 15, 21, 61, 74, 95] that exploit the lack of robustness in ML algorithms) are also out of scope as it is an orthogonal problem to our approach.

3 MOTIVATING EXAMPLE

We show how FreePart prevents vulnerabilities in the OpenCV framework [68] from corrupting critical data in a host application, OMRChecker [94] (auto-grader)[1]. It scans input OMR (Optical Mark Recognition) images using OpenCV, computing scores by comparing the recognized answers with the teacher's master answer. It grades multiple submissions and writes the results (*e.g.*, scores) to a `.csv` file.

Attack Scenario. A teacher uses OMRChecker [94] to grade multiple OMR images submitted by students, to obtain an output `.csv` file containing answers and scores recognized from the submissions. An attacker (*e.g.*, a malicious student) provides a maliciously crafted image that exploits a vulnerability [22] in `imread()` (an OpenCV API, A) to manipulate the grading process as shown in Fig. 1-(a) (1). Specifically, `template.QBlocks.orig` variable in OMRCheck (2) that defines the coordinates of answer mark areas in the submission image, represented as red boxes in Fig. 1-(b). The attacker exploits the vulnerability to corrupt the values of `template.QBlocks.orig`, changing the coordinates as shown in Fig. 1-(c), making the program incorrectly recognize students' responses (*e.g.*, all the images' *answer B* will be considered as *answer*

[1]Note that there are variants [84, 93] of OMRChecker (and one of them are registered in Google Play), sharing the same vulnerability.



(a) Code based API Isolation (b) Code based API and Data Isolation (c) Library based Isolation (for *the entire library*) (d) Library based Isolation (for *individual APIs*) (e) Isolation by FreePart

** (1) A and B represent vulnerabilities. (2) Red background processes mean that they are vulnerable. (3) Memory-based technique is omitted as it does not partition the process.*

Figure 2: Illustrations of Existing Techniques on the Motivating Example.

A). In addition, the attacker exploits a vulnerability in `imshow()` (B) to crash the application (3).

Goal. We aim to prevent the exploitation of the two OpenCV APIs `imread()` (A) and `imshow()` (B) from affecting two critical variables `template` (coordinates of the answer marks) and `OMRCrop` (the input OMR image).

3.1 Existing Techniques

There are three types of existing techniques: code-based, library-based, and memory-based isolation techniques. We illustrate how they would partition the program in Fig. 2-(a)~(d) for the motivation example (except for the memory-based technique that does not partition a program). Table 1 summarizes the effectiveness of existing techniques[2] and FreePart.

- **Code-based API Isolation [44] (Fig. 2-(a))** isolates APIs (but not data) by partitioning the host application's code. For our example scenario, there are three isolated processes. The first process runs the initialization code (*i.e.*, loading the `template`) and `imread()`. The second process executes `imshow()`. The third process runs the remaining APIs. Note that it requires users to manually annotate how to partition the program and what or where should the policies be applied, leading to unsystematic and ineffective protection. For example, the process running `imread()` also includes the `template` variable without protection, allowing the memory corruption attack, as shown in Fig. 1 (2). Worse, it *breaks* the host application's functionality as the isolated `imshow()` creates a GUI window, stored in a global variable, which is not accessible by APIs in other processes.
- **Code-based API and Data Isolation [13, 106] (Fig. 2-(b))** isolates *APIs and data* by partitioning the host application's code. Note that they require an accurate dependency analysis technique with annotations of variables from the user (*e.g.*, PtrSplit [50], PM [51] or SOAAP [36]) to effectively partition APIs and variables. Then, the partitioned variables and code are isolated in separate processes automatically. In this example, there can be 5 processes: 3 processes for isolating APIs (same as Fig. 2-(a)) and 2 processes to isolate `OMRCrop` and `template`, respectively. While it protects the data better than the code-based API isolation, it incurs *non-trivial overhead* due to the frequent access of the isolated `template` and `OMRCrop` in hot loops, causing a lot of IPCs (*e.g.*, more than 800 for each sample input). Note that [13, 106] aim to isolate privileged data (*e.g.*, secret keys) that are not frequently accessed, different from our scenario.
- **Library-based Isolation for Entire Library [10, 33, 63, 105] (Fig. 2-(c))** separates the host application's execution from the library's execution. It requires users to annotate library APIs' invocation sites, while it does not require sophisticated dependency analysis, making it more practical than other techniques. They run the *entire library code* in a *single* process. However, since all the library APIs exist in a single process, once an API is exploited, other APIs can be compromised, leading to data corruptions handled by the compromised APIs. For example, `warpPerspective()` returns a transformed image of an input image. If corrupted, it can manipulate transformed images that are input for other APIs, such as `morphologyEx()`. Moreover, [10] further reduces the overhead caused by IPCs by sharing variables via shared memory (*i.e.*, no IPCs for `OMRCrop`). However, it requires sophisticated data-dependency analysis and makes the shared variables vulnerable.
- **Library-based Isolation for Individual API [31] (Fig. 2-(d))** also separates the host application's execution and every library API's execution. Additionally, each API is isolated in a separate process, making it more secure than the former approach. Unfortunately, it incurs significant runtime overhead due to *IPCs on every API call*. The entire data of the API's arguments are transferred between processes on each API call. For instance, to process an image (1.7 MB) in our motivation example, there are 203 inter-process data transfers for 355 MB.
- **Memory-based Isolation [11]** protects the critical variables by assigning memory access permissions (*e.g.*, read-only). It requires sophisticated data dependency analysis to ensure the correctness

[2] We focus on the isolation/partitioning mechanism of existing techniques. Our example does not necessarily represent the full capability of them.

		Level of Security		Prevented	Isolated	API Isolation Granularity¶			# of	Performance¶¶
		Data*	APIs#	Attacks†	CVEs§	σ††	Min	Max	Processes‡	
Code-based	API[1]	○	◑	~~M~~ / C / D	1	47.9	1	84	3	●
	API & Data[2]	●	◑	M / C / D	2	37.3	0	84	5	◑
Library-based	Entire Library[3]	◑	○	~~M~~ / ~~C~~ / D	0	60.8	0	86	2	●
	Individual APIs[4]	●	●	M / C / D	2	0.1	0	1	87	○
	Memory-based[5]	◕	○	M / ~~C~~ / ~~D~~	0	-	86	86	1	●
	FreePart	◕	◕	M / C / D	2	32.4	0	75	5	●

●: Highly effective. ◕: Mostly effective. ◑: Less effective. ○: Not effective. (Each level elaborated in Section A.1.1) ∗: Level of security in protecting data. #: Level of security for APIs' execution. †: Three types of attacks: **M** for the memory corruption attack on critical data, **C** for the program code (*i.e.*, API function's code) manipulation (*i.e.*, code rewriting) attack, **D** for the denial of service attack by crashing the application. Crossed-out letters (*e.g.*, ~~M~~ and ~~C~~) mean that it failed to prevent the attacks. §: # of APIs with CVEs isolated. ¶: Granularity of processes w.r.t the # of APIs (Details in Section A.1.3). ††: Standard deviation of isolated APIs in different processes. ‡: # of processes required. ¶¶: Level of performance (Details in Section A.1.2). 1: Code-based API isolation (shown in Fig. 2-(a)). 2: Code-based API and Data isolation (shown in Fig. 2-(b)). 3: Library-based API isolation (shown in Fig. 2-(c)). 4: Library-based API and Data isolation (shown in Fig. 2-(d)). 5: Memory-based data isolation. **Green**, **orange**, and **red** background colors indicate desirable, moderate, and undesirable respectively.

Table 1: Effectiveness of Existing Techniques and FreePart.

in memory protection. It does not create additional processes for isolation. However, since APIs' execution is not isolated, a denial-of-service attack is possible. Also, it does not protect APIs from being compromised.

> **FreePart.** We provide practical solutions for protecting data and APIs with low overhead. It is based on framework API-based partitioning and isolation with temporal memory access permission enforcement via framework API invocations.

3.2 Exploit Mitigation by FreePart

Fig. 2-(e) shows how FreePart partitions the host program in the motivation example. There are 5 processes where 4 processes run each of the different types of framework APIs. The vulnerabilities (Ⓐ and Ⓑ) reside in two different processes. The last process contains the two critical data to be protected and FreePart controls the memory access permissions to prevent data corruption from the exploit. Note that the last row of Table 1 shows the performance of FreePart.

Framework API Categorization and Isolation. FreePart runs 4 different types of APIs (*e.g.*, data loading, data processing, visualizing, and storing) separately so that the exploitation of each type's APIs is confined within an isolated process. In this example, we categorize 86 APIs as shown in Table 2. We present how we decide the four API types in Section 4.1 and how we automatically categorize them in Section 4.2. Fig. 2-(e) shows that Ⓐ and Ⓑ are isolated in separate processes.

Data Protection via Temporal Partitioning. Fig. 3 shows how FreePart monitors API calls at runtime and changes the access permissions of the data. The x-axis represents the time of the execution. The '*Framework State*' row shows how FreePart maintains the current state of execution based on API calls: the initial state is

Type	# APIs	Examples of APIs
Data Loading	3	`cv2.imread()`, `pd.read_csv()`[1], `json.load()`[1]
Data Processing	75	`cv2.GaussianBlur()`, `cv2.erode()`, `cv2.Canny()`, `cv2.warpPerspective()`, `cv2.morphologyEx()`, ...
Visualizing	6	`cv2.imshow()`, `cv2.moveWindow()`, `plt.show()`[1], ...
Storing	2	`cv2.imwrite()`, `plt.savefig()`[1]

1: Those framework APIs of `pd` (pandas [73]), `json`, and `plt` (Matplotlib [91]) require our hybrid analysis to categorize them.

Table 2: Framework APIs Categorized for the Motivating Example.

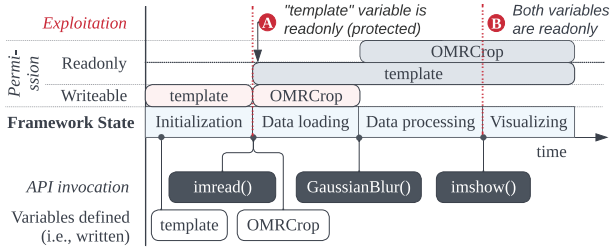


Figure 3: Timeline of API Calls and Data Protection.

"initialization," and the state changes on a framework API's call. For example, a call of `imread()` (a data-loading API) changes the current state to *data loading*. Similarly, calling a data processing API, *e.g.*, `GaussianBlur()`, changes the current state to *data processing*. As the state changes, FreePart changes the access permission of the variables defined in the previous state to *readonly*. Note that we require users to define the memory layout of a customized data structure (*e.g.*, buffer location and size of '`template`') to set the memory access permissions.

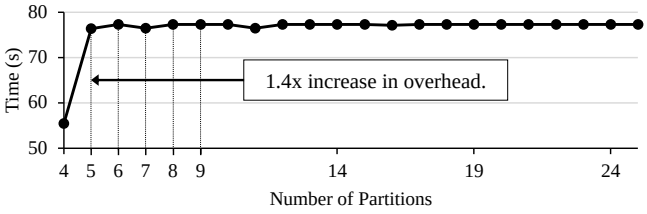


Figure 4: Average Runtime for Different Number of Partitions.

Name	Data Loading			Data Processing			Visualizing			Storing		
	Avg[1]	M[2]	T[3]	Avg[1]	M[2]	T[3]	Avg[1]	M[2]	T[3]	Avg[1]	M[2]	T[3]
OpenCV	0.6	1	1	0.2	1	1	0	0	0	0	0	0
TensorFlow	0.3	2	2	2.3	12	24	0	0	0	0	0	0
Pillow	0.4	2	2	0	0	0	0.5	1	1	0	0	0
NumPy	0.1	1	1	0.4	1	1	0	0	0	0	0	0
Total	**1.4**	5	6	**2.9**	14	26	**0.5**	1	1	**0**	0	0

1: Average # of vulnerable APIs in a single application. 2: Maximum # of vulnerable APIs in a single application. 3: Total # of vulnerable APIs across all 56 applications.

Table 3: Categorization of Vulnerable APIs in 56 Applications.

Mitigation of Memory Corruption and DoS. Fig. 2-(e) shows that the vulnerable function `imread()` is executed in a separate process (*i.e.*, data loading process), protecting the `template` variable from being corrupted. Denial-of-service (DoS) attacks by A and B are mitigated as they only crash the data loading and visualizing processes, respectively. In addition, as shown in Fig. 3-A, FreePart makes the `template` variable *readonly* on and after the `imread()` call, protecting the variable.

Mitigation of Code Manipulation. Code manipulation (*e.g.*, code rewriting) requires a malicious payload to change the memory permission via system calls such as `mprotect()`. It is mitigated as FreePart only allows system calls required for framework APIs isolated in each process. Note that each isolated process runs *the same type* of framework APIs (typically requiring a similar set of system calls), enabling the effective system call restriction.[3]

Choice of Four Partitions. While increasing the number of partitions (*i.e.*, fine-grained partitioning) may increase security, it would incur performance overhead. To understand the trade-off of the security and performance regarding the partitions, we conduct experiments for a different number of partitions. From the 4 partitions (*i.e.*, data loading, data processing, storing, and visualizing), we try to increase the number of partitions from 4 to 25, resulting in more than 155K combinations. Specifically, there are 23 APIs and we create 7,750 different partitions (randomly created) for each number of partitions from 5 to 25.

Fig. 4 shows that the average overhead increases 1.4 times when the number of partitions is increased from 4 to 5. This is because there are two functions, *i.e.*, `cv.rectangle` and `cv.putText` (both without CVEs), in hot-loop (*i.e.*, frequently executed), sharing a substantial amount of data. When they are separated into different partitions, a substantial overhead occurs. Note that the two APIs in this example do not have known vulnerabilities, meaning that the finer-grained partitioning does not offer better security in this example *practically*. We elaborate on more details in Section A.1.4.

4 DESIGN

Workflow. As shown in Fig. 5, FreePart takes the source code of the target host program and the frameworks used by the host program. It first gets a list of all the framework APIs used in the program. Then, it runs a hybrid analysis (*i.e.*, static and dynamic combined) to categorize framework APIs. Finally, we hook the identified framework APIs and objects used in the APIs at runtime to implement communications between the isolated processes and enforce the protections.

4.1 Studies for FreePart's Design

We present two studies that obtain insights for designing our framework-based partitioning and isolation.

Study 1: Usage of Framework APIs. We manually analyze 56 popular programs (selected by the number of stars of their GitHub repositories) using data processing frameworks to check whether the execution of framework APIs can be used to infer temporal partitions for the isolation. As shown in Fig. 6, we observe that all the analyzed applications follow the *data loading*, *data processing*, and *visualizing* or *storing* workflow.[4] Specifically, a program typically loads an input file, runs an algorithm on the data, and then presents visualizations or stores the results in files. Some programs, such as video processing programs, repeat the data loading and processing. Note that an output of a component is an input of the next component, and the next component *only reads* the input, supporting our data protection via temporal partitioning.

> The pipeline style pattern of framework APIs' execution motivates the framework-based API partitioning. The observation that components only read the input motivates us to make the memory of the previous component readonly when a new type of framework API is called.

Study 2: API Types and Vulnerabilities. We study 241 publicly available CVEs (from August 2018 to February 2022) related to data processing frameworks: TensorFlow (172 CVEs), Pillow (44 CVEs), OpenCV (22 CVEs), and NumPy (3 CVEs). For each CVE, we identify which task in Fig. 6 triggers or is being affected by the vulnerability.

Fig. 7 shows the categorized result with types of vulnerabilities (*e.g.*, DoS and unauthorized file/memory access). Vulnerabilities exist across all the API types, while the majority of them are in the data loading and data processing APIs. To better understand the security implications of the vulnerabilities, we investigate them further. First, we find vulnerabilities in utility functions such as CVE-2019-16249 and CVE-2019-15939 can be exploited during the execution of multiple types of APIs, potentially affecting multiple processes. Second, we realize that even if there are many vulnerable APIs in a particular API type, each application only uses a few of them (*e.g.*, data processing type uses 2.9 vulnerable APIs on average).

[3] Techniques running APIs and the application code together (Fig. 2-(a)~(b)) or diverse types of APIs together (Fig. 2-(c)~(d)) require diverse system calls to be allowed, making system call restriction ineffective.

[4] Note that programs without GUI may not use visualizing APIs

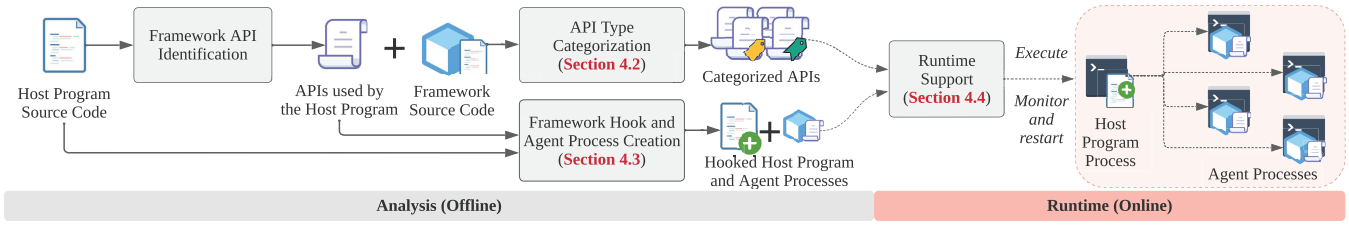


Figure 5: Workflow of FreePart (From the Offline Analysis to the Online Runtime Enforcement).

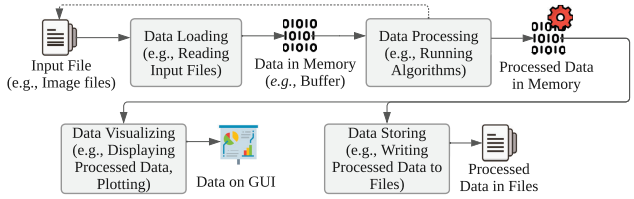


Figure 6: Pipeline Pattern of Data Processing.

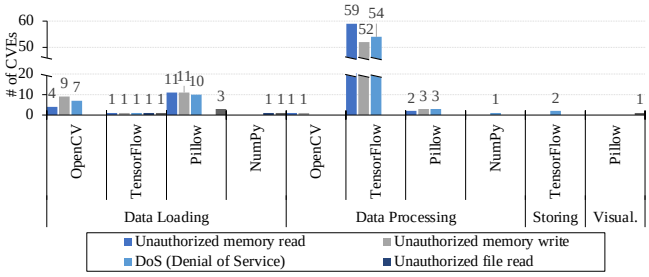


Figure 7: CVEs Categorized by Types of Vulnerabilities.

In other words, for individual applications, we only need to isolate the vulnerable APIs *actually used*, instead of vulnerable but *unused* APIs. Table 3 shows how many vulnerable APIs are used in the 56 applications in our study. Observe that each data loading or processing agent process only includes 2~3 vulnerable APIs on average.

> Our study on the CVEs shows that vulnerabilities are all across the four types of APIs. In addition, for a single application, there is only a handful of vulnerable APIs in each agent process as shown in Table 3.

4.2 Automated API Type Categorization

We develop a hybrid analysis (*i.e.*, static and dynamic analyses combined) to automatically categorize framework APIs based on the data flow patterns during the execution due to the large number of APIs (*e.g.*, OpenCV and Caffe have 1,405 APIs [71] and 224 APIs [27] respectively).

Definitions. To facilitate discussion, we introduce a few formal definitions describing operations that cause data transfers in Fig. 8. Data read and write operations are modeled by R(S_{src}) and W(S_{dst}, S_{src}), with S_{src} and S_{dst} holding the definition of data source and destination, respectively. Storage S defines the origins of data. Specifically, MEM represents the memory, GUI means variables and objects that are relevant to GUI (Graphical User Interface): `g_windows` and `cvNamedWindow()`. FILE and DEV represent a file (in the file system) and a device such as a camera, respectively.

$$\begin{array}{lll} \textit{Operation} & O ::= & S_{dst} = \mathrm{R}(S_{src}) \mid \mathrm{W}(S_{dst}, S_{src}) \\ \textit{Storage} & S ::= & \textsc{mem} \mid \textsc{gui} \mid \textsc{file} \mid \textsc{dev} \mid \ldots \end{array}$$

Figure 8: Definitions for Data Flow Patterns.

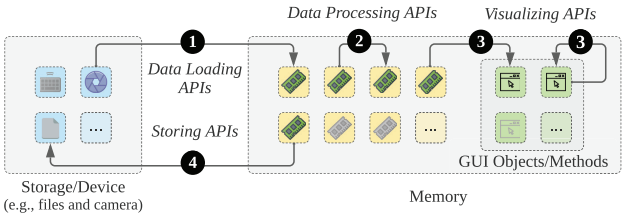


Figure 9: Patterns of Dependencies for Framework APIs.

4.2.1 Data Flow Patterns. Fig. 9 shows data flow patterns that correspond to the four different types of framework APIs.

1. *Data Loading APIs:* If an API uses system calls to load data from storage or devices (*e.g.*, retrieving an image from a camera) to memory, it is categorized as the data loading API. Hence, APIs containing 'W(MEM, R(FILE or DEV))' operations are the data loading type.
2. *Data Processing APIs:* Most APIs running algorithms are only reads/writes memory is categorized as the data processing type. APIs that have 'W(MEM, R(MEM))' operations but not others are the data processing type.
3. *Visualizing APIs:* Visualizing APIs display the content to the user (*e.g.*, `imshow()`). We identify them by detecting APIs that access the GUI-relevant objects (*e.g.*, `g_windows`). Specifically, APIs that have one of the following operations are categorized as the visualizing type: 'W(GUI, R(MEM))', 'W(MEM, R(GUI))', and 'R(GUI)'.
4. *Storing APIs:* An API that stores data to the storage or device (*i.e.*, W(FILE or DEV, R(MEM)) operation) is categorized as the storing type.

Memory Copy via Files. APIs can use storage as a space to copy data. For instance, `tf.keras.utils.get_file()` in TensorFlow

consists of three operations: (1) downloading from the network (MEM_{buf1} = R($\text{DEV}_{network}$)), (2) storing the downloaded data to a file (W(FILE_{tmp}, MEM_{buf1})), and (3) reading the file content to a buffer (MEM_{buf2} = R(FILE_{tmp})). The file operations on FILE_{tmp} are to pass the data from MEM_{buf1} to MEM_{buf2}. Hence, we reduce the operations to "MEM_{buf2}=R(MEM_{buf1})", making it a data loading API.

4.2.2 Hybrid Analysis. FreePart runs a static analysis first to identify the data flow pattern in a framework API. To handle data flows missed by static analysis (*e.g.*, APIs having dynamically allocated objects and indirect calls), we use dynamic analysis.

Static Analysis. We identify system calls that read/store data (*e.g.*, `read()` and `write()`) to indicate the data flows between storage/devices and the memory. For memory reads/writes, we focus on assignment statements (*e.g.*, '`x = y`'). APIs that do not have data loading/storing system calls are categorized as data processing APIs. For visualizing APIs, FreePart searches statements or functions accessing GUI objects or invoking GUI relevant functions. Note that our static analysis might have false positives and negatives due to language constructs such as indirect calls and pointers. To solve the problem, we use dynamic analysis.

Dynamic Analysis. We obtain test cases based on frameworks' examples and test cases [27, 69, 77, 78], covering most of the APIs[5] in the framework (Table 11 in Appendix A.3). We measure the test runs' code coverage by using Coverage.py [64] and llvm-cov [20]. For programs that require user interactions, we use Monkey Tools [8] to randomly generate user interactions. Table 4 shows a few examples of the framework APIs with categories (More APIs can be found in Appendix A.4).

	Type	Functions / Classes
OpenCV	DL[1]	`imread()`, `cvLoad()`, `VideoCapture()`, `readOpticalFlow()`, ...
	DP[2]	`CascadeClassifier()`, `cvtColor()`,`equalizeHist()`, ...
	V[3]	`setWindowTitle()`, `getMouseWheelDelta()`, `imshow()`, ...
	S[4]	`imwrite()`, `writeOpticalFlow()`, `VideoWriter()`, ...
Caffe	DL[1]	`ReadProtoFromTextFile()`, `ReadProtoFromBinaryFile()`, ...
	DP[2]	`Forward()`, `Backward()`, `CopyTrainedLayersFrom()`, ...
	S[4]	`hdf5_save_string()`, `WriteProtoToTextFile()`, ...
PyTorch	DL[1]	`load()`, `hub.load()`, `utils.model_zoo.load_url()`, ...
	DP[2]	`argmax()`, `tensor()`, `nn.Conv2d()`, `combinations()`, ...
	S[4]	`save()`, `utils.tensorboard.writer.SummaryWriter()`, ...
TensorFlow	DL[1]	`image_dataset_from_directory()`, `utils.get_file()`, ...
	DP[2]	`nn.conv3d()`, `nn.avg_pool()`, `nn.max_pool()`, ...
	S[4]	`preprocessing.image.save_img()`, `Model.save_weights()`, ...

* Caffe, PyTorch, and TensorFlow do not have visualizing type APIs hence omitted.
1: Data Loading. 2: Data Processing. 3: Visualizing. 4: Storing.

Table 4: API Type Categorization Example.

Execution Partitioning with Framework APIs. With the categorized APIs, FreePart partitions the execution of the host application and enforces memory permissions[6]. Fig. 10-(a) shows a slightly modified version of a facial recognition program [67] which is the host application. It initializes a `VideoCapture` object to read image frames from a camera (line 1) and a classifier for facial recognition (line 3). It also loads user profiles that contain personal information related to the facial recognition models (line 4). The program starts a loop that fetches frames from the camera (line 5). For each frame, it runs the facial recognition algorithm (`cascade.detectMultiScale()`) (lines 7~10). At line 12, it sends out the detection results to another server. Finally, it shows the current frame on the screen (line 14) and writes the current frame to a file if the 's' key is pressed (line 16). The program terminates if the 'q' key is pressed (lines 17~19).

Framework APIs with the same type are grouped and annotated by the circled letters: Ⓛ, Ⓟ, Ⓢ, and Ⓥ for the data loading, data processing, storing, and visualizing respectively. Fig. 10-(b) and (c) are the modified source code of the library interface and implementation of the agent process.

Type-neutral Framework APIs. We observe that there are framework APIs that do only memory-to-memory operations and are frequently used together with different types of APIs. Unfortunately, when such two APIs with different types also share the data, it may cause substantial overhead via IPCs between the agent processes. For instance, `cvtColor()` is a data processing API which is frequently used together with `detectMultiScale()` (object detection algorithm; data processing API) to create a gray scale image and `imshow()` (a visualizing API). Utility APIs such as `cvCreateMemStorage()` and `cvAlloc()` are also used together with different types of APIs. Since their semantics are dependent on the calling context, we consider the type of such APIs is also flexible or neutral. To this end, we call them type neutral APIs, and their types are determined by the types of other APIs used together. To this end, we run them in an existing agent process together. For example, if `cvtColor()` is used right after the data loading process API `imread()`, we run the `cvtColor()` on the data loading agent process. Similarly, if `cvtColor()` is used in the middle of the data processing APIs (*e.g.*, `GaussianBlur()`), we run it in the data processing agent process.

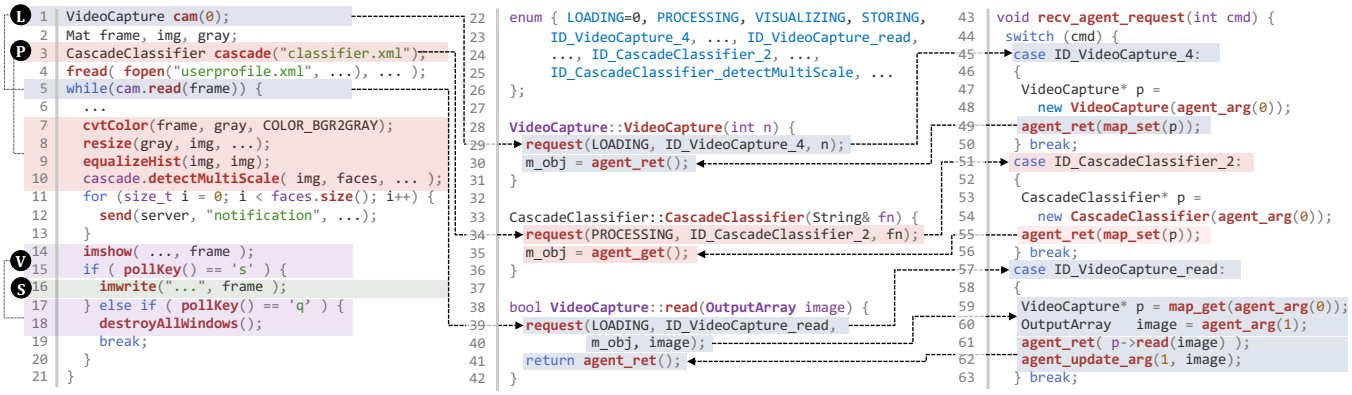
4.3 Framework Hook and Agent Process Creation

FreePart automatically instruments the categorized APIs and data structures used in the APIs (as an argument or return). For framework APIs, we use the `LD_PRELOAD` trick [59] to hook the APIs since all the frameworks we support use dynamic libraries by default[7]. Each hooked API function then intercepts the API calls and runs the API's code in an agent process (*i.e.*, essentially making a remote procedure call to the agent process). Our API hooking seamlessly redirects the program execution without affecting the correctness of the program (*i.e.*, we do not change input/output of the APIs except for enforcing security policies). Note that we assume symbols of the framework and library functions are presented, which is typical. FreePart also hooks data structures and objects' methods by redefining them. For instance, `Mat` is a frequently used data structure for image data in OpenCV. The image data in `Mat` are stored in a heap buffer. Since a framework API takes or returns

[5] Note that the APIs that are not covered by our test cases (*i.e.*, outside of the 'most' APIs) are not used by any of our evaluated programs in Section 5.

[6] We leverage `mprotect()` to change memory permissions.

[7] For frameworks using static libraries, we hook APIs in source code.



(a) Target Program (b) Hooked Interfaces (Functions and Objects) (c) Request Handler in an Agent Process

Figure 10: Example API Categorization, Interface Hooking, and Agent Processes.

a reference (*i.e.*, address) of a `Mat` object (*not the heap buffer*), we implement a deep copy of the object when its reference is passed to or returned by APIs.

FreePart as RPC. FreePart's API hooking essentially implements a remote procedure call (RPC). Specifically, FreePart implements the "*exactly-once*" semantic, meaning that a request to execute an API on an agent process will be delivered to the agent process and executed *exactly* once.

4.3.1 Hook Interfaces and Agent Processes. Fig. 10-(b) shows an example of how FreePart hooks methods and connects them to the agent processes. Note that it is automatically instrumented according to the definition of the original framework APIs. Arrows between Fig. 10-(a) and (b) indicate control flow transfers from the host program to the hooked interfaces. `request()` (at lines 29, 34, and 39) sends a framework API execution request with arguments to the agent process.[8] Lines 22~26 define constant identifiers used in the program. `request()`'s arguments include (1) the API type (*e.g.*, `LOADING` for the data loading), (2) the id of the target function, and (3) arguments of the target function (including its object reference). Fig. 10-(c) shows the source code of an agent process. The agent process accepts requests, runs the requested APIs (with the command id through the `cmd` and the arguments retrieved through `agent_arg()` and `agent_arg()`), and copies the results back to the host application (via `agent_ret()` at lines 49, 55, and 61~62).

4.3.2 Lazy Data Copy for Optimizing IPCs. We observe that in typical data-processing programs, results returned by a framework API are often *immediately used* by another framework API. FreePart leverages the observation to reduce IPCs by *directly copying the data between the agent processes*, without going through the host application's process. We call this optimization '*Lazy Data Copy (LDC)*,' which reduces the unnecessary data copy, *only if* data loaded by a framework API are *directly* fed to another framework API in a different agent process. LDC essentially *postpones* data copy operations until the data are dereferenced by a concrete agent process. Then, LDC allows the agent process to copy data *directly*.

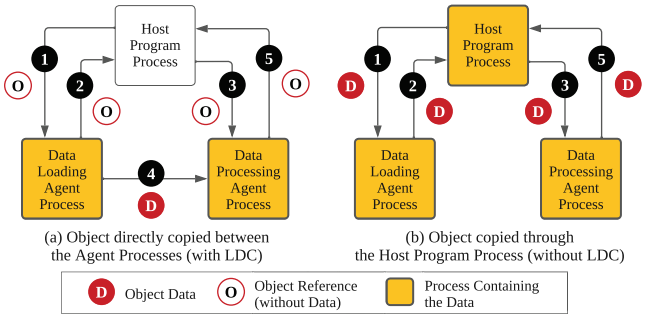


Figure 11: Inter Process Data Flow with the Lazy Data Copy.

Fig. 11-(a) and (b) show examples of data flows with and without the LDC, respectively.

1. The host program process calls `imread()`, passing all the arguments of the function to the data processing agent process. With LDC, it passes the *references* of the objects. Without LDC, it passes *all the data* of the objects. The *reference* only contains the *origin*[9] of the object's data.
2. The data loading agent loads the data and returns it to the host program process. After the function, all the arguments (as they might be modified) and the return values are copied back. Without LDC, the *entire data* of objects are passed, even though the host program process does *not* access the data. With LDC, the *references* are passed.
3. The host program calls `equalizeHist()` (*i.e.*, data processing API), which sends a request to run the API with the *loaded object* in the data processing agent process. Again, the references are passed with LDC while the entire data of objects are passed without LDC.
4. The data processing agent process runs the `equalize Hist()` API, beginning to access the data. With LDC, since only the references were passed, it copies the data *directly* from the

[8] We implement IPC (Inter-Process Communication) between processes using shared memory. It uses ring buffers and futex for synchronization.

[9] The agent process's process id (PID) and the identifier of the buffer (*e.g.*, a hash value of the buffer's address) that contains the data.

APIs / Objects	Required System Calls
`CascadeClassifier::load()`	`openat, close, brk, fstat, read, lseek,`
`VideoCapture::VideoCapture()`	`openat, close, ioctl, mmap`
`VideoCapture::read()`	`brk, ioctl, select`

(a) Required System Calls for OpenCV in Fig. 9's Program

```
openat, close, brk, fstat, read, lseek, ioctl, mmap, select
```

(b) Data Loading Agent Process for Fig. 9's Program.

Figure 12: Obtaining Required System Calls.

data loading process at this time. Without LDC, the data were already passed at ❸.

❺ The resulting object is returned to the host process. Without LDC, the data of the resulting objects are copied back.

4.4 Runtime Support

FreePart's runtime support consists of a loader and a dynamic library. The loader is a standalone program that initializes the host and agent processes. The dynamic library hooks framework APIs and data objects, initializes the IPC channels, monitors the execution and enforces security policies.

4.4.1 Restricting System Calls. Each framework API does not need to access all the system calls. For example, to run a data-loading framework API in OpenCV, around 5~6 system calls are needed on average [30] (Syscalls Per API). Hence, FreePart restricts system calls unnecessary for executing framework APIs. For example, Fig. 12-(a) shows the system calls required for the data loading APIs used by the program shown in Fig. 10. Then, we create an *allowed system call list* by taking the union of required system calls for all framework APIs within an agent process as shown in Fig. 12-(b).

Identifying Required System Calls. We use the hybrid analysis described in Section 4.2.2 to identify required system calls for each framework API. Note that our hybrid analysis has a high code coverage of the framework APIs (as reported in Table 11), resulting in high-confidence results.

Overlapping System Calls Between APIs. On average, we find that 6 system calls are required per API. Among them, 4 system calls (*e.g.*, `openat`, `fstat`, `brk` and `read` in the data loading agent process) commonly appear across the APIs of the same type. FreePart allows the superset of system calls used by APIs in the agent process.

System Call Restrictions. We use `seccomp-BPF` [72] to permit system calls in the allowed list. The allowed list ensures that the system call restriction also prevents system object interface attacks (*e.g.*, restricting `shm_open` and `mprotect` to manipulating shared memory or re-writing memory permissions). To protect FreePart from attackers reconfiguring `seccomp-BPF` to tamper with the system call restriction, we use `PR_SET_NO_NEW_PRIVS` which prevents configuration changes. In addition, system calls, such as `ioctl`, require an additional restriction on their arguments because they can access diverse devices (*e.g.*, a camera to retrieve images or a network device for communication) depending on their arguments. For such APIs (*e.g.*, `ioctl`, `connect`, `select`, and `fcntl`), FreePart checks their file descriptors to ensure they operate only on the designated files.

System Calls Required During the Initialization. A few security-critical system calls such as `mprotect` and `connect` are used by framework APIs but *only in their first execution.* Our manual analysis result shows that they are only required for the initialization. For instance, `connect` is used only once during the first execution of a visualizing API to initialize a socket to communicate to a GUI subsystem and `mprotect` is used to load library modules on its first execution. Since those system calls are *not* required *after* the first execution of the APIs, FreePart first executes all the framework APIs and then restricts them afterwards.

4.4.2 Restarting Agent Processes. Server programs (*e.g.*, web servers) often prioritize availability over security, automatically restarting the process when it crashes. For them, FreePart provides functionality to restart agent processes so that FreePart does not break the original workflow. If a user prioritizes security over availability, one can opt out of it. Note that since a separate process restarts the agent processes, a compromised agent processes cannot influence the restarting functionality.

FreePart as RPC. In terms of RPC, FreePart implements the restarting agent process via "*at-least-once*," meaning that for crashed processes, FreePart may re-execute APIs multiple times. Note that this is acceptable as most framework APIs are stateless. There are a few stateful APIs. For them, we periodically store the states. We elaborate it on Appendix Section A.2.4.

4.4.3 Deriving and Enforcing Memory Access Permissions. As shown in Fig. 3 (in Section 3) FreePart enforces memory access permissions (*e.g.*, read-only) when the framework changes its state. Specifically, it requires a user to annotate a data structure to be protected, including the functions that *create* the data (*e.g.*, the constructor of a class if the data is an object) and *access* the data (*e.g.*, read/get methods of the data object). FreePart supports the definitions of such functions for popular data structures in supported frameworks such as `Mat` in OpenCV. However, for the user-defined data structures, the definitions should be manually provided.

Given the definitions, FreePart's runtime infers the current framework's state by monitoring which type of framework API is invoked. The current framework state is essentially reflecting the last framework API's type. There are 5 framework states: Initialization, Data Loading, Data Processing, Visualizing, Data Storing. The initialization state represents the initial state before any framework API's invocation.

When the framework state is changed (*e.g.*, the program starts to call a data processing API, after calls to data loading APIs), it enforces the memory access permissions of all the *data objects defined in the previous state read-only*. For example, if the state just changed to the data processing from the data loading, all the data objects defined (*i.e.*, created/allocated) during the data loading state will become read-only.

5 EVALUATION

Implementation. The prototype of FreePart is implemented with 3,829 SLOC. We use LLVM [90] to categorize framework APIs by analyzing their data flow patterns and to instrument the frameworks written in C/C++. In addition, we use PyCG [80] to conduct

Vuln. Type	CVE IDs	Vuln. Samples[1]	Type
Unauthorized Mem. Write	CVE-2017-12604, CVE-2017-12605, CVE-2017-12606, CVE-2017-12597	1, 9, 10, 12	DL[2]
Remote Code Execution	CVE-2017-17760	1, 7, 10, 12	DL[2]
	CVE-2019-5063, CVE-2019-5064	1, 9, 10	DP[3]
Denial-of-Service (DoS)	CVE-2017-14136, CVE-2018-5269	1, 7, 9, 10, 12	DL[2]
	CVE-2019-14491, CVE-2019-14492, CVE-2019-14493	1, 9, 10	DP[3]
	CVE-2021-29513	21, 23	DP[3]
	CVE-2021-29618	23	DP[3]
	CVE-2021-37661	21, 22, 23	DP[3]
	CVE-2021-41198	20, 22	DP[3]

1: Sample IDs having the vulnerability. 2: Data Loading. 3: Data Processing.

Table 5: List of CVEs used for Evaluation.

static analysis on Python programs. Our runtime support is written in C++.

Experimental Setup. All the experiments were done on a machine with Intel i7-9750H, 2.6GHz, 32GB RAM, and 64-bit Ubuntu 18.04 with a GeForce RTX 2060. We use a commercial off-the-shelf (COTS) system without any modifications.

Program Selection. We search open-source repositories to find 30 popular applications (*i.e.*, by the number of GitHub stars [32]) using the data processing frameworks that we support: OpenCV, Caffe, PyTorch, and TensorFlow. From the searched result, we sort them by the number of source lines of code (SLOC) to filter trivial projects. In addition, if there exist multiple similar programs (*e.g.*, there are many facial recognition applications with a similar code structure), we only select one project that has versatile functionalities and ignore similar applications. To this end, we choose 23 programs (out of 30) as shown in Table 6: 9 for OpenCV (and OpenCV based), 3 for Caffe, 10 for PyTorch, and 4 for TensorFlow. Some applications use multiple frameworks together (*e.g.*, Face_classification [5] uses OpenCV and Keras together).

Vulnerability Selection and Attack Construction. We searched the CVE database to find vulnerabilities on OpenCV, Caffe, Torch (for PyTorch), and TensorFlow reported in the last five years that are also used by the selected 23 programs. We successfully reproduce 18 vulnerabilities, that form the set of CVEs we use in this evaluation. Table 5 shows the vulnerability type and the selected CVEs that we use in this evaluation. The third column shows sample program IDs (shown in Table 6) that have the vulnerability (hence affected). The last column shows the different types of vulnerable framework API (also representing which agent process it will belong to). Then, we create attacks by constructing exploits for the selected vulnerabilities, either by improving already provided Proof-of-Concept (PoC) exploits or by creating our own from scratch. We use Metasploit [79] to create malicious payloads (*e.g.*, ROP payloads and shellcodes).

Correctness of FreePart. We apply FreePart to all 23 programs. For all programs, we use their respective test suites (linked each on our website [30]) publicly available as inputs to test the functionality. We observe that all the test cases are correctly executed, without detecting any attacks incorrectly (*i.e.*, no false positives). We also conduct 8 attacks by reproducing 18 CVEs (shown in Table 5) and 6 attacks used in case studies (Section 5.4 and Section A.7) on the 23 programs. They are all successfully mitigated, meaning that we did not observe false negatives. We further manually inspect the API categorization of the 23 programs and confirmed that all partitioned APIs were correctly categorized conforming to the data flow patterns outlined in Section 4.2.1.

5.1 API Type Categorization Result

The 7th~14th columns of Table 6 show the numbers of framework APIs categorized to each type. "Unique" shows the unique number of APIs used in the application and "Total" shows the number of API call instances of each type.

Categorized APIs. The data loading type has the smallest number of framework APIs used in the programs. However, since they are the interface functions that directly handle untrusted user inputs, they are major targets of attacks (and have many vulnerabilities). The data processing type has the most framework APIs (both Unique and Total). Note that in the data processing APIs, the total numbers are significantly larger than unique numbers, suggesting there are multiple call sites of a single framework API. Our manual inspection reveals that those programs have many duplicated code snippets to implement multiple optimized versions for different workloads.

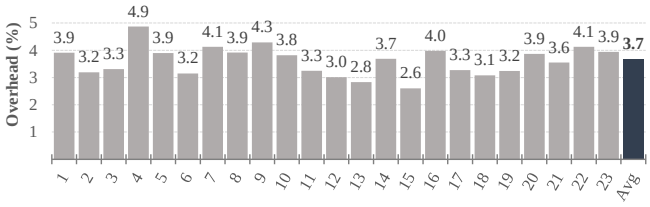


Figure 13: Normalized Runtime Overhead of FreePart.

5.2 Runtime Performance Overhead

We measure the runtime performance overhead of FreePart with the 23 programs in Table 6. Specifically, we run the original programs and the FreePart protected programs with various workloads and compare the performance results. We collect the workloads as follows. First, for applications that provide demo and test workloads [65, 66, 70], we use them. Second, we additionally use non-trivial image/video data sets (144GB) collected from ImageNet [24] and text data (a few MBs) for image/video and text processing applications [62].

Fig. 13 shows the results: the average overhead is 3.68%. The hybrid analysis for each application took 1 hour on average. We observe that our low overhead is largely due to partitioning dependent APIs together in the same process (*i.e.*, avoiding data copies on each RPC).

Lazy Data Copy. To understand the effectiveness of our lazy data copy, we measure the overhead of FreePart without lazy data copy support, which is 9.7% for 23 programs on average. We observe that

ID	Framework	Name	Lang.	SLOC	Size	Loading*		Processing+		Visualizing		Storing		Description
						Uniq†	Total	Uniq†	Total	Uniq†	Total	Uniq†	Total	
1	OpenCV[1]	Face_classification [5]	Python	7,082	280K	4	4	5	10	4	4	1	1	Face, emotion, gender detection
2	OpenCV	FaceTracker [48]	C/C++	3,012	588K	2	5	19	99	3	3	3	6	Real-time deformable face tracking
3	OpenCV	Face_Recognition [2]	Python	3,205	14.8M	1	8	5	26	3	15	2	3	Face recognition application
4	OpenCV	lbpcascade_anime [62]	Python	6,671	224K	1	1	4	4	3	3	1	1	Image classification/object detection
5	OpenCV	EyeLike [92]	C/C++	742	44K	5	5	21	100	4	18	1	2	Webcam based pupil tracking
6	OpenCV	Video-to-ascii [42]	Python	483	48K	4	7	2	2	0	0	1	1	Plays videos in terminal
7	OpenCV	Libfacedetection [111]	C/C++	14,016	8.8M	4	6	14	62	4	4	1	1	Library for face detection
8	OpenCV	OMRChecker [94]	Python	1,797	6.2M	2	4	42	88	4	5	1	1	Grading application
9	Caffe[2]	EmoRecon [87]	Python	1,773	53K	6	10	11	32	5	6	1	1	Real-time emotion recognition
10	Caffe[2]	Openpose [14]	C/C++	459,373	6.8M	10	12	44	171	2	2	0	0	Real-time person keypoint detection
11	Caffe[2]	MTCNN [46]	Python	425	12.9K	1	1	11	18	2	2	0	0	MTCNN face detector
12	PyTorch[3]	SiamMask [102]	Python	39,999	1.4M	2	9	19	103	4	10	2	11	Object tracking and segmentation
13	PyTorch	CycleGAN-pix2pix [116]	Python	1,963	7.64M	5	7	50	103	0	0	1	2	Image-to-image translation
14	PyTorch	FAIRSEQ [28]	Python	39,800	5.9M	8	19	20	65	0	0	4	4	Sequence modeling toolkit
15	PyTorch	PyTorch-GAN [26]	Python	6,199	31.1M	3	105	41	1,747	0	0	1	37	PyTorch implementation of GANs
16	PyTorch	YOLO-V3 [96]	Python	2,759	1.98M	3	9	68	254	3	3	2	6	PyTorch implementation of YOLOv3
17	PyTorch	StarGAN [18]	Python	740	2.07M	1	2	32	105	0	0	1	4	PyTorch implementation of StarGAN
18	PyTorch	EfficientNet [57]	Python	2,554	2.48M	4	8	37	86	0	0	2	2	PyTorch implementation of EfficientNet
19	PyTorch	Semantic-Seg. [115]	Python	3,699	5.53M	2	2	136	304	0	0	1	3	Semantic segmentation/scene parsing
20	TensorFlow	DCGAN-TensorFlow [88]	Python	3,142	67.4M	3	6	54	137	0	0	1	1	TensorFlow implementation of DCGAN
21	TensorFlow	See in the Dark [16]	Python	610	836K	1	8	31	244	0	0	2	10	Learning-to-See-in-the-Dark (CVPR'18)
22	TensorFlow	CapsNet [39]	Python	679	486K	1	8	43	108	0	0	4	6	TensorFlow implementation of CapsNet
23	TensorFlow	Style-Transfer [56]	Python	731	11M	3	4	37	61	0	0	3	5	Add styles from images to any photo

∗: Data Loading. +: Data Processing. †: Unique. 1: Uses OpenCV (main) and Keras (secondary) APIs. 2: Uses Caffe (main) and OpenCV (secondary) APIs.
3: Uses PyTorch (main) and OpenCV (secondary) APIs.

Table 6: Applications used for Evaluation.

about 95% of the data copy operations are lazy data copies, indicating the target applications mostly have data flows only between the framework APIs. More details can be found in Appendix A.5.

5.3 Security Analysis on Attack Scenarios

We present an analysis of FreePart under typical attack scenarios: (1) data exfiltration and (2) data corruption attacks.

The data exfiltration attack represents a typical scenario of stealing sensitive information (*i.e.*, information leak). We assume a powerful attacker who is capable of identifying the exact memory addresses of the buffer containing sensitive data. Given a memory address of a buffer to leak, the attack aims to send the critical information to attacker-controlled servers via network APIs such as `send()`.

The data corruption attack corrupts critical data in the program, such as outcomes of the algorithms or other metadata used in the program (*e.g.*, sensitive user profiles). We assume that the attacker already knows the exact memory addresses to compromise (*i.e.*, memory addresses of buffers containing sensitive information). Then, the attacker leverages the RCE vulnerability (*e.g.*, CVE-2019-5063) to overwrite critical data.

We launch the two attacks to all vulnerable programs as shown in Table 5. We then analyze what information can be stolen or what damages can be made by the attacks.

Analysis of Data Exfiltration. For all the programs we evaluate, we find that most sensitive information stays within the target program process. Since all the vulnerabilities we tested are in the data loading or data processing processes, attacks could not access sensitive information that exists in the target program process. In the data loading process, if a target program processes inputs from multiple users (*e.g.*, a server program that detects objects in pictures provided by remote users), other users' inputs can be considered sensitive. In the data processing process, the processed outcomes (*e.g.*, facial recognition results of a specific person) might be sensitive.

While the attacks can access those, both data loading and data processing processes do not allow system calls that can write data to the disk or other devices (*e.g.*, `write` or `send`), meaning that it is difficult to send the stolen information to outside. Table 7 shows a few allowed system calls for each API type. Note that all the API types are for OpenCV.

Type	Allowed system calls
Loading (43)	`bind`, `fstat`, `futex`, `getcwd`, `getpid`, `listen`, `mkdir`, `openat`, `recvfrom`, ...
Processing (22)	`getrandom`, `gettimeofday`, `open`, `openat`, `read`, `close`, `clock_gettime`, ...
Visualizing (56)	`access`, `connect`, `eventfd2`, `futex`, `getuid`, `lseek`, `select`, `sendto`, ...
Storing (27)	`accept`, `close`, `dup`, `exit`, `lstat`, `mkdir`, `umask`, `uname`, `unlink`, ...

Table 7: System Calls Allowed for Each API Type.

Analysis of Data Corruption Attack. If the attack happens in the data loading process, the attack may corrupt *previous inputs* of the program. However, since it is already passed to the data

processing process and processed, corrupting previous inputs has practically no impact. An attacker may corrupt the data, which will be passed to the next process (*i.e.*, to another API type). However, this is essentially providing a crafted input. In the data processing process, an attacker may corrupt the data currently processing. However, it is difficult in practice, as the corrupted values are likely to be overwritten during the algorithm, meaning that the attacker has limited controllability. In addition, an attacker may target ML models and configurations on the memory. However, we observe that compromising them in the middle of computation mostly leads to wrecking the result.

5.4 Case Study

5.4.1 Autonomous Object Tracking Drone. We use FreePart to mitigate attacks on an autonomous drone project [17] that keeps tracking an object using an object recognition technique via a camera attached to the drone. The drone follows the recognized object as it moves. The program fetches images from the camera and uses `imread()` (which has the vulnerability) to load and process. We prepare two attacks: (1) a DoS attack that crashes the drone program and (2) a data corruption attack that modifies the speed of the drone by overwriting a configuration variable.

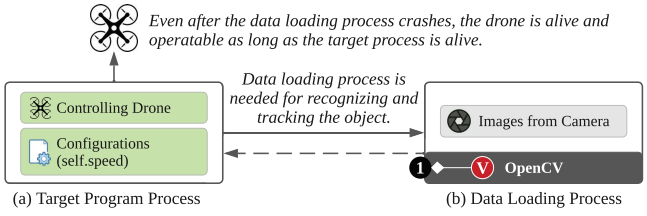


Figure 14: FreePart Protected Autonomous Drone [17].

DoS Attack. An attacker exploits CVE-2017-14136 and CVE-2019-14491 vulnerabilities, which can crash the entire drone program. Without FreePart, the entire program crashes, and the drone will *halt its operation and fall to the ground*. As shown in Fig. 14, with FreePart, the crash happens in the data loading process (in `imread()`), crashing the process. Note that even if FreePart does not restart the crashed process, the target program process is still alive, *making the drone alive*. Before it restarts the process, it may not handle new images from the camera. However, the user can safely land the drone as all other functionalities (*e.g.*, controlling the drone) are not affected. With the process restarting, FreePart can seamlessly mitigate this DoS attack, while the drone might be a little sluggish due to the restarting of the isolated process.

Data Corruption Attack. An attacker exploits CVE-2017-12606 vulnerability to corrupt a specific configuration variable that defines the speed of the drone. Specifically, the drone's speed is stored in the `self.speed` variable. The default speed is 0.3, and changing it to '−0.3' will make the drone move in the opposite direction (*i.e.*, not following the object but moving away from the object). Without FreePart, the attacker can access the variable and modify its value.

With FreePart, the exploitation is contained within the data loading process (❶ in Fig. 14), while the variable `self.speed` exists in the target program process.

5.4.2 Information Leak in an Image Viewer. MComix3 [53] is an image viewer program which is forked from the MComix [60] project. The program has a menu listing recently loaded files' names. An attacker aims to leak the recent file names which might be sensitive information. They are stored in `self._window.uimanager.recent` and `Gtk::RecentManager` which is a part of the GTK library (*i.e.*, a GUI framework).

An attacker can use CVE-2020-10378 to read the variables. Then, it can send the information through network APIs such as `connect()` and `send()`. With FreePart, the attack will fail because the exploitation happens in the data loading process while the `self._window.uimanager.recent` exists in the target program process, and `Gtk::RecentManager` exists in the visualizing process. Attempts to access the variable in the target program process and in the vizualizing process (❷ in Fig. 15) fail. Moreover, sending the information through the network will be prevented by the system call restriction.

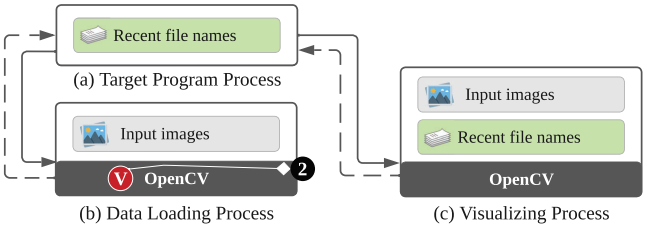


Figure 15: FreePart Protected MComix3 Program.

6 DISCUSSION

Restoring States of Crashed Process. When a process crashes, FreePart intentionally decides not to restore values of variables in the crashed process after restarting the crashed process. This is because the crash might be caused by an attack with a malicious payload. Note that this may cause state discrepancies between processes after the isolated process is recovered from a crash.

Impact of API Miscategorization. If our hybrid analysis (in Section 4.2.2) fails to identify data flows (false negative) or incorrectly detects bogus data flows (false positive) described in Fig. 9, framework APIs can be miscategorized. Miscategorization would cause two major consequences. First, when a vulnerability in a miscategorized API is exploited, it can have access to sensitive data that should not be allowed access. Second, miscategorized APIs can cause many unnecessary IPC communications as they frequently access variables that do not exist in the same process. This is because FreePart uses a separate process to run each type of framework API.

Impact of Intra-Process Attacks. While partitioning using FreePart reduces the attack surface, it is still vulnerable to intra-process attacks. For instance, an attacker can arbitrarily execute malicious code in the vulnerable process. However, FreePart reduces the attack surface by limiting capabilities (*e.g.*, by employing syscall restriction), restricting the privileges of the compromised API (*i.e.*, same memory-access privileges as the API type). In addition, an attacker can attempt control flow hijacking attacks. However, one can employ CFI [98] and debloating solutions [108] to mitigate them.

On the other hand, an attacker can also influence other APIs or memory (stack and heap) within the compromised process. For this, one can employ other intra-process partition techniques [38, 41] to the agent processes to mitigate or reduce the effects of the attack. Note that, applying security techniques to isolated processes is an orthogonal problem.

Partitioned Processes and Multi-threading. FreePart executes with fives processes (1 host program process and 4 agent processes). Each partitioned process has its independent stack and heap (mitigating all memory corruption attacks *across partitions* while corrupting a compromised agent process's local stack is possible and is our limitation). For multi-threading processes, each thread will have its own set of four agent processes, hence avoiding race conditions.

7 RELATED WORK

Software Fault Isolation (SFI). Program partitioning and SFI are closely related to FreePart. [101] is one of the earliest software-based fault isolation approaches, which instruments a target program to detect unsafe memory accesses. Since then, SFI has been applied to various targets, including OS kernels [113], system applications [12], web applications [25, 40, 110], and mobile apps [7, 75], to partition the existing system into sub-components by functionality. [105] needs developers to manually write wrappers for the target untrusted library to partition and isolate libraries while FreePart does it automatically to prevent attackers who can compromise the user mode program's memory. Existing works have used diverse partitioning approaches, such as static analysis and instrumentation [25], compiler modifications [12], runtime primitive (*e.g.*, sandboxing) [7, 105, 110], and additional language primitive support [40, 75]. A key difference is that existing approaches focus on partitioning a target program, while FreePart focuses on partitioning from frameworks.

Automated Program Dependence Analysis. Program dependency analysis aims to identify program statements and data that are required for the correct execution of other statements. Researchers have used static analysis [13], dynamic analysis [45, 47, 107], and hybrid methods [54] to perform dependency analysis for program partitions. In addition, source code-based techniques [19, 112] can leverage annotations to help with the analysis. Program slicing techniques [3, 104, 114] are proposed to identify a set of program statements required to execute a certain functionality. There are slicing techniques for concurrent programs [29], source code locality identification [43], and clustering-based program component decomposing [58]. For example, BCD [43] uses static analysis to extract the code locality, data references and function calls. Bunch [58] uses clustering to decompose applications. Advanced techniques can further optimize FreePart's API type identification and partitioning.

Intra-process Isolation. Recently, intra-process isolation techniques have been proposed to isolate a part of the program from the remainder to enhance security. Note that these techniques are orthogonal to our approach and can be adapted to complement FreePart. Recent techniques such as Hodor [38] and SeCage [55] utilize virtualization to enable different memory views for each program part, isolating data and code. Meanwhile, other techniques [34, 82, 97, 100] utilize PKU-based memory isolation technique to protect critical data. They focus on enhancing memory security for data on heap memory, while they may not be effective in preventing attacks on stack memory or non-memory attacks.

Endokernel [41] attempts to solve the problem by mapping their virtual machine abstraction to system-level objects. It provides programmable security abstractions that can be used to monitor and secure the system against low-level attacks with low overhead. In addition, Jenny [81] employs the system call filter rules necessary for protecting the former PKU-based isolation domains. Note that Endokernel and Jenny provide practical security primitives that can be used to enhance FreePart. However, to apply them, one needs to partition the target application first, which may require sophisticated data-dependency analysis. Our technique provides a practical partitioning approach for data-processing applications without requiring complex dependency analysis.

8 CONCLUSION

We present FreePart which mitigates the impact of vulnerabilities in data processing frameworks onto its host applications. It leverages framework-based software partitioning to identify API types of a target program and isolate each framework API belonging to a certain API type into a separate process. FreePart effectively confines the exploited execution, preventing it from escaping from each partitioned process and damaging the system. Our experiments on 23 applications on four widely used frameworks (OpenCV, Caffe, PyTorch, and TensorFlow) and 18 vulnerabilities show that it effectively prevents attacks with negligible overhead (3.68%).

ACKNOWLEDGMENT

We thank the anonymous referees for their constructive feedback. The authors gratefully acknowledge the support of NSF (1908021, 1916499, 2145616, and 1955719), and DARPA (N6600120C4020). This research was also supported by a Google Faculty Fellowship and a gift from Cisco Systems. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] R.P. Abbott, Lawrence Livermore Laboratory, Institute for Computer Sciences, and Technology. *Security Analysis and Enhancements of Computer Operating Systems: The RISOS Project, Lawrence Livermore Laboratory.* U.S. Department of Commerce, National Bureau of Standards, 1976.

[2] Adam Geitgey. The world's simplest facial recognition api for Python and the command line, 2020. https://github.com/ageitgey/face_recognition.

[3] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, 25(6):246–256, June 1990.

[4] Ionut Arghire. Serious Vulnerabilities Patched in OpenCV Computer Vision Library, 2020. https://www.securityweek.com/serious-vulnerabilities-patched-opencv-computer-vision-library.

[5] Octavio Arriaga, Matias Valdenegro-Toro, and Paul Plöger. Real-time convolutional neural networks for emotion and gender classification. *arXiv preprint arXiv:1710.07557*, 2017.

[6] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018*, July 2018.

[7] Elias Athanasopoulos, Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. Nacldroid: Native code isolation for android applications. In *European Symposium on Research in Computer Security*, pages 422–439. Springer, 2016.

[8] Autoit. Autoit. https://www.autoitscript.com/site/.

[9] Krishnakumar Balasubramanian and Saeed Ghadimi. Zeroth-order (non)-convex stochastic optimization via conditional gradient and gradient updates. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 3459–3468, Red Hook, NY, USA, 2018. Curran Associates Inc.

[10] Markus Bauer and Christian Rossow. Cali: Compiler-assisted library isolation. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, page 550–564, New York, NY, USA, 2021. Association for Computing Machinery.

[11] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 309–322, USA, 2008. USENIX Association.

[12] Ajay Brahmakshatriya, Piyus Kedia, Derrick P McKee, Deepak Garg, Akash Lal, Aseem Rastogi, Hamed Nemati, Anmol Panda, and Pratik Bhatu. Confllvm: A compiler for enforcing data confidentiality in low-level code. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[13] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, page 5, USA, 2004. USENIX Association.

[14] Zhe Cao, T. Simon, Shih-En Wei, and Yaser Sheikh. Realtime multi-person 2d pose estimation using part affinity fields. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1302–1310, 2017.

[15] Nicholas Carlini and D. Wagner. Towards evaluating the robustness of neural networks. *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57, 2017.

[16] cchen156. Learning-to-See-in-the-Dark. https://github.com/cchen156/Learning-to-See-in-the-Dark.

[17] Mohamed Chaabane. Autonomous-flight-of-the-drone-AR.Drone-1.0. https://github.com/MedChaabane/Autonomous-flight-of-the-drone-AR.Drone-using-OpenCV.

[18] Yunjey Choi, Minje Choi, Munyoung Kim, Jung-Woo Ha, Sunghun Kim, and Jaegul Choo. StarGAN: Unified Generative Adversarial Networks for Multi-Domain Image-to-Image Translation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[19] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. *SIGOPS Oper. Syst. Rev.*, 41(6):31–44, October 2007.

[20] Chris Lattner and Vikram Adve. llvm-cov tool shows code coverage information for programs, 2020. https://llvm.org/docs/CommandGuide/llvm-cov.html.

[21] Kenneth T. Co, Luis Muñoz González, Sixte de Maupeou, and Emil C. Lupu. Procedural noise adversarial examples for black-box attacks on deep convolutional networks. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 275–289, New York, NY, USA, 2019. Association for Computing Machinery.

[22] CVE. CVE-2017-12597. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-12597.

[23] CVE. CVE-2019-19781 - Vulnerability in Citrix Application Delivery Controller, Citrix Gateway, and Citrix SD-WAN WANOP appliance, 2019. https://support.citrix.com/article/CTX267027.

[24] Jia Deng, R. Socher, Li Fei-Fei, Wei Dong, Kai Li, and Li-Jia Li. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 00, pages 248–255, 06 2009.

[25] Adam Doupé, Weidong Cui, Mariusz Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. dedacota: toward preventing server-side xss via automatic code and data separation. pages 1205–1216, 11 2013.

[26] Erik Linder-Norén. PyTorch implementations of Generative Adversarial Networks., 2020. https://github.com/eriklindernoren/PyTorch-GAN.

[27] Evan Shelhamer. Caffe Deep learning framework. https://caffe.berkeleyvision.org/.

[28] FaceBook. Facebook AI Research Sequence-to-Sequence Toolkit written in Python., 2020. https://github.com/pytorch/fairseq.

[29] Moreno Falaschi, Maurizio Gabbrielli, Carlos Olarte, and Catuscia Palamidessi. Slicing concurrent constraint programs. In *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 76–93. Springer, 2016.

[30] FreePart. FreePart Code Release, 2020. https://github.com/freepart2022/FreePart-22.

[31] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. Enclosure: Language-based restriction of untrusted libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 255–267, New York, NY, USA, 2021. Association for Computing Machinery.

[32] GitHub. GitHub Stars, 2020. ttps://stars.github.com/.

[33] Google. Google/sandboxed-api: Generates sandboxes for c/c++ libraries automatically. https://github.com/google/sandboxed-api.

[34] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. EPK: Scalable and efficient memory protection keys. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 609–624, Carlsbad, CA, July 2022. USENIX Association.

[35] guanshuicheng. Invoice, 2021. https://github.com/guanshuicheng/invoice.

[36] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean application compartmentalization with soaap. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1016–1031, New York, NY, USA, 2015. Association for Computing Machinery.

[37] Ankit Gupta. What is OpenCV and why is it so popular?, 2019. https://medium.com/analytics-vidhya/what-and-why-opencv-3b807ade73a0.

[38] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process isolation for High-Throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, Renton, WA, July 2019. USENIX Association.

[39] Huadong Liao. CapsNet. https://github.com/naturomics/CapsNet-Tensorflow.

[40] Casen Hunger, Lluís Vilanova, Charalampos Papamanthou, Yoav Etsion, and Mohit Tiwari. Dats-data containers for web applications. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 722–736, 2018.

[41] Bumjin Im, Fangfei Yang, Chia-Che Tsai, Michael LeMay, Anjo Vahldiek-Oberwagner, and Nathan Dautenhahn. The endokernel: Fast, secure, and programmable subprocess virtualization. *CoRR*, abs/2108.03705, 2021.

[42] Joel Ibaceta. Video to Ascii. https://github.com/joelibaceta/video-to-ascii.

[43] Vishal Karande, Swarup Chandra, Zhiqiang Lin, Juan Caballero, Latifur Khan, and Kevin Hamlen. BCD: Decomposing Binary Code Into Components Using Graph-Based Clustering. In *13th ACM ASIA Conference on Information, Computer and Communications Security*, Songdo, Korea, June 2018.

[44] Douglas Kilpatrick. Privman: A library for partitioning applications. In *2003 USENIX Annual Technical Conference (USENIX ATC 03)*, San Antonio, TX, June 2003. USENIX Association.

[45] Dohyeong Kim, Yonghwi Kwon, William N. Sumner, Xiangyu Zhang, and Dongyan Xu. Dual execution for on the fly fine grained execution comparison. *SIGARCH Comput. Archit. News*, 43(1):325–338, March 2015.

[46] kuangliu. MTCNN with pycaffe. https://github.com/kuangliu/pycaffe-mtcnn.

[47] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. Ldx: Causality inference by lightweight dual execution. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 503–515, New York, NY, USA, 2016. Association for Computing Machinery.

[48] Kyle McDonald. Real time deformable face tracking in C++ with OpenCV 3.

[49] Lindsey O'Donnell. Chinese Hackers Exploit Cisco, Citrix Flaws in Massive Espionage Campaign, 2020. https://threatpost.com/chinese-hackers-exploit-cisco-citrix-espionage/154133/.

[50] Shen Liu, Gang Tan, and Trent Jaeger. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2359–2371, 2017.

[51] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. Program-mandering: Quantitative privilege separation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1023–1040, 2019.

[52] Tao Liu, Zihao Liu, Qi Liu, Wujie Wen, Wenyao Xu, and Ming Li. Stegonet: Turn deep neural network into a stegomalware. In *Annual Computer Security Applications Conference*, pages 928–938, 2020.

[53] WJ Liu. MComix3. https://github.com/multiSnow/mcomix3.

[54] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1607–1619, New York, NY, USA, 2015. Association for Computing Machinery.

[55] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1607–1619, New York, NY, USA, 2015. Association for Computing Machinery.

[56] Logan Engstrom. Style transfer. https://github.com/lengstrom/fast-style-transfer.

[57] lukemelas. A PyTorch implementation of EfficientNet, 2020. https://github.com/lukemelas/EfficientNet-PyTorch.

[58] Spiros Mancoridis, Brian Mitchell, Yih-Farn Chen, and Emden Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. 04 1999.

[59] Linux manual page. Linux Programmer's Manual dynamic linker/loader. https://man7.org/linux/man-pages/man8/ld.so.8.html.

[60] MComix. MComix: GTK+ comic book viewer. https://sourceforge.net/p/mcomix/wiki/Home/.

[61] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: A simple and accurate method to fool deep neural networks. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2574–2582,

2016.
[62] Nagadomi. A Face detector for anime/manga using OpenCV, 2018. https://github.com/nagadomi/lbpcascade_animeface.
[63] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 699–716. USENIX Association, August 2020.
[64] Ned Batchelder. Code coverage measurement for Python, 2020. https://github.com/nedbat/coveragepy.
[65] OpenCV. Extra data for the OpenCV library . https://github.com/opencv/opencv_extra.
[66] OpenCV. Test code for the OpenCV library . https://github.com/opencv/opencv/tree/master/modules/core/test.
[67] OpenCV. OpenCV object detection example. https://github.com/opencv/opencv/blob/master/samples/python/tutorial_code/objectDetection/cascade_classifier/objectDetection.py.
[68] OpenCV. OpenCV Project. https://opencv.org/.
[69] OpenCV. Performance testing in OpenCV. https://github.com/opencv/opencv/wiki/HowToUsePerfTests.
[70] OpenCV. Samples for the OpenCV library. https://github.com/opencv/opencv/tree/master/samples.
[71] OpenCV. Open Source Computer Vision, 2020. https://docs.opencv.org/4.1.0/d2/d75/namespacecv.html.
[72] Linux Kernel Organization. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/v5.0/userspace-api/seccomp_filter.html.
[73] The pandas development team. pandas-dev/pandas: Pandas, February 2020.
[74] Nicolas Papernot, Patrick D. McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 372–387, 2016.
[75] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 71–72, 2012.
[76] PyTorch. PyTorch. https://pytorch.org/.
[77] PyTorch. Training a Classifier. https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html.
[78] Suriyadeepan Ramamoorthy. torchtest. https://github.com/suriyadeepan/torchtest, 2019.
[79] Rapid7. Metasploit, 2020. https://www.metasploit.com/.
[80] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. Pycg: Practical call graph generation in python. In *43rd International Conference on Software Engineering*, ICSE '21, 2021.
[81] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing syscalls for PKU-based memory isolation systems. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 936–952, Boston, MA, August 2022. USENIX Association.
[82] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient In-Process isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694. USENIX Association, August 2020.
[83] Scikit-learn. Scikit-learn: Machine Learning in Python, 2020. https://scikit-learn.org/stable/.
[84] Shreyas. OMR Checker. https://github.com/letssolvetogether/OMRChecker.
[85] SolarWinds MSP. RCE: Remote Code Execution Explained, 2019. https://www.solarwindsmsp.com/blog/remote-code-execution.
[86] Steve Zurier. TensorFlow revokes support for YAML because of arbitrary code execution vulnerability, 2021. https://www.scmagazine.com/analysis/devops/tensorflow-revokes-support-for-yaml-because-of-arbitrary-code-execution-vulnerability.
[87] Sushant. Real-Time Facial Emotion Recognition with Convolutional Neural Nets, 2017.
[88] Taehoon Kim. DCGAN Tensorflow. https://github.com/carpedm20/DCGAN-tensorflow.
[89] TensorFlow. Tensorflow: An end-to-end open source machine learning platform, 2020. https://www.tensorflow.org/.
[90] The LLVM Foundation. The LLVM Compiler Infrastructure Project. https://llvm.org/.
[91] The Matplotlib Development team. Matplotlib - Visualization with Python, 2022. https://matplotlib.org/.
[92] Trishume. A webcam based pupil tracking implementation., 2019.
[93] Udayraj Deshmukh. An android application for validating images of OMR sheets before they are sent for processing, 2019. https://github.com/Udayraj123/AndroidOMRHelper.
[94] Udayraj Deshmukh. Grade exams fast and accurately using a scanner or your phone, 2020. https://github.com/Udayraj123/OMRChecker.
[95] Jonathan Uesato, Brendan O'Donoghue, Aäron van den Oord, and Pushmeet Kohli. Adversarial risk and the dangers of evaluating against weak attacks. In *ICML*, 2018.
[96] Ultralytics. YOLOv3 in PyTorch. https://github.com/ultralytics/yolov3.
[97] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, Santa Clara, CA, August 2019. USENIX Association.
[98] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 927–940, New York, NY, USA, 2015. Association for Computing Machinery.
[99] Vidita V Koushik. Uncovering critical vulnerabilities in real-time computer vision library, OpenCV, 2020. https://www.secpod.com/blog/opencv-buffer-overflow-vulnerabilities-jan-2020/.
[100] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. You shall not (by)pass! practical, secure, and fast pku-based sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 266–282, New York, NY, USA, 2022. Association for Computing Machinery.
[101] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, 1993.
[102] Qiang Wang, Li Zhang, Luca Bertinetto, Weiming Hu, and Philip HS Torr. Fast online object tracking and segmentation: A unifying approach. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1328–1338, 2019.
[103] Jinpeng Wei and Calton Pu. Tocttou vulnerabilities in unix-style file systems: An anatomical study. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, page 12, USA, 2005. USENIX Association.
[104] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press, 1981.
[105] Yongzheng Wu, Sai Sathyanarayan, Roland HC Yap, and Zhenkai Liang. Codejail: Application-transparent isolation of libraries with tight program interactions. In *European Symposium on Research in Computer Security*, pages 859–876. Springer, 2012.
[106] Yongzheng Wu, Jun Sun, Yang Liu, and Jin Song Dong. Automatically partition software into least privilege components using dynamic data dependency analysis. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 323–333. IEEE, 2013.
[107] Yongzheng Wu, Jun Sun, Yang Liu, and Jin Song Dong. Automatically partition software into least privilege components using dynamic data dependency analysis. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, page 323–333. IEEE Press, 2013.
[108] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. Subdomain-based generality-aware debloating. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 224–236, New York, NY, USA, 2021. Association for Computing Machinery.
[109] xming521. CTAI, 2020. https://github.com/xming521/CTAI.
[110] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53(1):91–99, January 2010.
[111] Shiqi Yu. an open source library for CNN-based face detection in images., 2020.
[112] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3):283–328, August 2002.
[113] Weijuan Zhang, Xiaoqi Jia, Shengzhi Zhang, Rui Wang, and Peng Liu. Running os kernel in separate domains: A new architecture for applications and os services quarantine. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 219–228, Dec 2018.
[114] Xiangyu Zhang and Rajiv Gupta. Cost effective dynamic program slicing. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, page 94–106, New York, NY, USA, 2004. Association for Computing Machinery.
[115] Bolei Zhou, Hang Zhao, Xavier Puig, Tete Xiao, Sanja Fidler, Adela Barriuso, and Antonio Torralba. Semantic understanding of scenes through the ade20k dataset. *International Journal on Computer Vision*, 2018.
[116] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networkss. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, 2017.

A APPENDIX

A.1 Motivating Example Data

A.1.1 Level of Security in Table 1. In Section 3, we compare FreePart to existing techniques in terms of security. We use the metric

"Security Level" in Table 1, which we define as follows. Specifically, we rank them from 'Highly effective' (●) to 'Not effective' (○). The detailed rubrics are in Table 8.

A.1.2 Performance in Table 1. In Table 9, we show the performance breakdown of FreePart and related approaches. Specifically, the overall performance presented in Table 1 is computed by considering the total runtime overhead (in seconds) shown in the third column of Table 9. The number of IPCs and the amount of transferred data between processes are presented in the first and second columns.

	Data			
	●	◉	◑	○
Memory-corruption on `OMRCrop` is mitigated	☑	☑	☑	☐
Memory-corruption on `template` is mitigated	☑	☑	☐	☐
Memory permissions enforced to avoid unauthorized writes on `OMRCrop`	☑	☑	☐	☐
Memory permissions enforced to avoid unauthorized writes on `template`	☑	☑	☐	☐
`OMRCrop` memory is not shared with APIs	☑	☐	☐	☐
`template` memory is not shared with APIs	☑	☐	☐	☐
	API			
	●	◉	◑	○
Code-rewriting attack on other API code mitigated	☑	☑	☑	☐
Vulnerable `imread()` isolated	☑	☑	☑	☐
Vulnerable `imshow()` isolated	☑	☑	☐	☐
APIs distributed in 5 or more processes	☑	☑	☐	☐
APIs isolated in individual processes (≥86 processes)	☑	☐	☐	☐

Table 8: Ruberic for Level of Security of Data and APIs.

		# of IPC‡	Data# (GB)	Time* (seconds)	Overall†
Code-based	API[1]	169	0.1	54.3	●
	API and Data[2]	6,854	21.9	88.8	◑
Library-based	Entire Lib.[3]	12,411	0.0	54.9	●
	Individual APIs[4]	12,411	42.7	121.8	○
	Memory-based[5]	0	0.0	54.1	●
	FreePart	12,411	0.4	55.6	●

‡: Total number of IPC calls. #: Total data transferred between processes. †: Overall overhead. ● : Low Overhead (<10% increase). ◑ : Moderate Overhead (>50% increase). ○ : High Overhead (>100% increase). ∗: Total time taken. 1: Code-based API isolation (shown in Fig. 2-(a)). 2: Code-based API and Data isolation (shown in Fig. 2-(b)). 3: Library-based isolation for the entire library (shown in Fig. 2-(c)). 4: Library-based isolation for individual APIs (shown in Fig. 2-(d)). 5: Memory-based data isolation.

Table 9: Overhead of Existing Techniques and FreePart.

A.1.3 Granularity of Isolation of Existing Techniques and FreePart. Table 10 shows the granularity of API isolation of existing techniques mentioned in Section 3 and FreePart. Each value in the table shows the number of framework APIs in each partition (or isolated process).

A.1.4 Partitioning Beyond Four Partitions. In this section, we elaborate on our analysis of finer-grained partitioning beyond the four partitions used in FreePart. In particular, we try to reason by further partitioning the existing four partitions, particularly the data processing agent, which contains 74 framework APIs, as shown in Table 10. To split the partition (with 74 APIs in data processing APIs), there are more than 1.8e+22 ways. We tried (*i.e.*, subsampled) 155K combinations of partitions by randomly choosing APIs for the new partitions. To this end, we find an average overhead increase of *16 times* in the worst case. This is because there are two frequently executed APIs: `cv.rectangle` and `cv.putText` in the data processing agent.[10] If the new partitioning separates the APIs into different partitions, they cause significant overhead.

Note that they two follow the pipeline style pattern of data processing that FreePart leverages, avoiding such overhead.

		Process Number					
		1	**2**	**3**	**4**	**5**	**≥ 6**
Code-based	API[1]	1	1	84	-	-	-
	API and Data[2]	1	1	84	0	0	-
Library-based	Entire Lib.[3]	0	86	-	-	-	-
	Individual APIs[4]	1	1	1	1	1	1
	Memory-based[5]	86	-	-	-	-	-
	FreePart	3	75	6	2	0	-

1: Code-based API isolation (shown in Fig. 2-(a)). 2: Code-based API and Data isolation (shown in Fig. 2-(b)). 3: Library-based isolation for the entire library (shown in Fig. 2-(c)). 4: Library-based isolation for individual APIs (shown in Fig. 2-(d)). 5: Memory-based data isolation.

Table 10: API Isolation Granularity.

A.2 Extended Design

A.2.1 Handling Complex Control Structures. There are many challenges in applying application-based partitioning to target programs because of complex control structures. We provide a few examples of them.

Try-Catch Structure. Fig. 16-(a) shows a code snippet taken from the `readResponse()` function mentioned in Section 3 (Motivation). Note that there is a `try-catch` structure that surrounds the statements from lines 4 to 8. A desirable partitioning in this example is to partition the `show()` function at line 8 from other statements because `show()` is a GUI relevant function and others (*e.g.*, `resize_util()`) calls data processing functions (*e.g.*, `cv2.resize()`).

Fig. 16-(b) shows two functions that are partitioned (`partition1()` and `partition2()`). Lines 4~7 are partitioned to `partition1()` (lines 18~21) and line 8 is partitioned to `partition2()` (line 35). There are also IPC functions added to communicate between the partitioned programs (lines 22~24, 33~34, and 36), highlighted in gray.

Observe that the `try-catch` statements are copied to both `partition1()` and `partition2()` (denoted by ❶ and ❷ respectively). Otherwise, runtime exceptions in a partitioned function will not be preserved, breaking the target program's functionality.

[10] They are used to annotate different answers in an input image.

```
1  def readResponse(...):
2    try:
3      ...
4      img = resize_util(img, ...)
5      morph = img.copy()
6      ...
7      if(config.showimglvl>=4):
8        show("morph1",morph,0,1)
9      ...
10   except Exception as e:
11     exc_type, exc_obj, exc_tb = sys.exc_info()
12     fname = os.path.split(...)[1]
13     print("Error from readResponse: ", e)
14     print(exc_type, fname, exc_tb.tb_lineno)
```

(a) Original Program

```
15 def partition1(...):
16   try:
17     ...
18     img = resize_util(img, ...)
19     morph = img.copy()
20     ...
21     if(config.showimglvl>=4):
22       IPC.enqueue(morph)
23       IPC.signal(sig_partition2)
24       IPC.waitfor(sig_partition2_done)
25     ...
26   except Exception as e:
27     exc_type, exc_obj, exc_tb = sys.exc_info()
28     fname = os.path.split(...)[1]
29     print("Error from readResponse: ", e)
30     print(exc_type, fname, exc_tb.tb_lineno)

31 def partition2(...):
32   try:
33     IPC.waitfor(sig_partition2)
34     IPC.dequeue(morph)
35     show("morph1",morph,0,1)
36     IPC.signal(sig_partition2_done)
37     ...
38   except Exception as e:
39     exc_type, exc_obj, exc_tb = sys.exc_info()
40     fname = os.path.split(...)[1]
41     print("Error from readResponse: ", e)
42     print(exc_type, fname, exc_tb.tb_lineno)
```

(b) Partitioned Program (Application based Partitioning)

Figure 16: Challenges in Application-based Partitioning.

FreePart handles this by catching and redirecting exceptions that happen in the framework APIs. Note that since FreePart does not change the target programs' code, exceptions in the target programs' code are not affected.

Loop Structure and Function Call Chain. Fig. 17-(a) shows another code snippet from the `readResponse()` function which is a `for` loop calling `saveOrShowStacks()`. Note that `saveOrShowStacks()` should be partitioned because it can eventually call framework APIs that belong to two different API types (data processing and Visualizing).

This example shows two challenges. First, to partition a target function that calls framework APIs belonging to different process, one needs to analyze all the functions called in the target function until all the framework APIs belong to a single API type, hence single process. Second, when a target function for partitioning is inside a loop, the partitioned code should preserve the loop structure.

1. A function calling APIs belong to different API types: Since to partition the line 3, we analyze and partition statements in `saveOrShowStack()`. Since `resize_util()` calls a data processing API (`cv2.resize()`) and `saveImg()` invokes a storing process API (`cv2.imwrite()`), we partition the second function, `saveImg()`, and isolate it in another process. This is done in `saveOrShowStacks_partition1()` and `saveOrShowStacks_partition2()` (lines 21~23 and 26~30 respectively).

However, `show()` calls multiple framework APIs that belong to different API types: Visualizing (*e.g.*, `imshow()`) and data processing (`resize_util()`). To partition the function, one needs to analyze statements in `show()` as well. `show_partition3()` and `show_partition4()` show the resulting partitioned function.

2. Partitioning code in a loop: Note that there are two `while` statements at lines 27 and 42 that do not exist in the original program. Those are added because of line 2. Specifically, in the original program `saveOrShowStacks()` is expected to be executed multiple times (since it is in a loop), meaning that APIs called by the function will be invoked multiple times as well. If the partitioned functions do not have the `while` loop, it has to create a new process every time, which will cause significant performance overhead. The loop essentially makes the partitioned process alive to handle multiple requests if framework APIs are called in a loop in the original program.

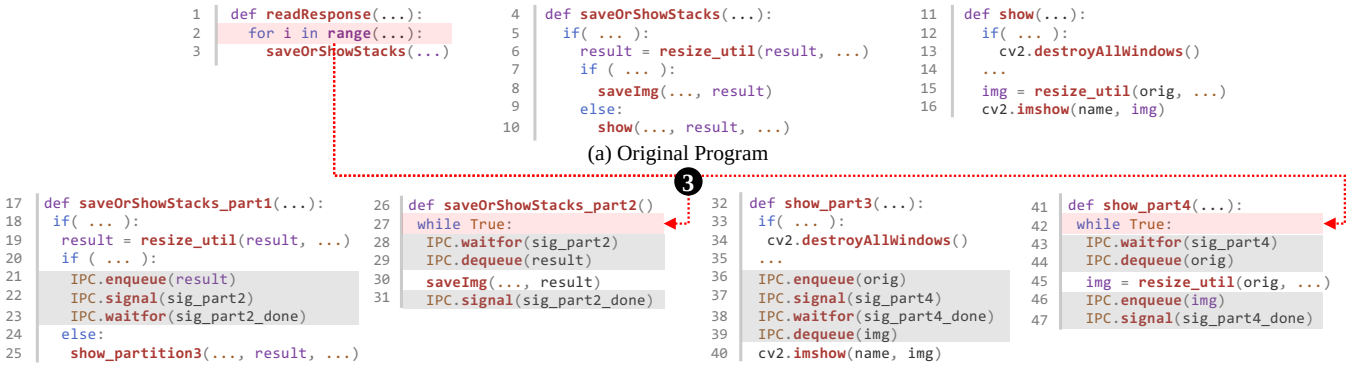
Observe that the analysis must remember whether a target statement for partitioning might be in a loop (*i.e.*, any callers of the statement are in the loop). The analysis is challenging because it needs to analyze all the caller functions of the statement. Worse, it is more challenging if one of the callers is an indirect function call (via a function pointer).

A.2.2 Performance. We observe that the framework instrumentation approach is resulting in less overhead as the instrumenting of a target program often ends up creating more duplicated data values across the processes. In typical cases, it also causes more inter-process data transfers between the processes. While it can be mitigated by a precise and accurate program analysis technique, state-of-the-art program dependency analysis techniques and implementations are difficult to handle complex real-world applications.

A.2.3 Scalability. Even if the instrumentation is possible, target applications, in practice, are written in various languages, and it is challenging to develop instrumentation tools for all such diverse languages.

A.2.4 Restoring States for Restarted Agent Processes. When FreePart restarts an agent process, it needs to retain states for stateful APIs (*i.e.*, APIs that behave differently depending on states stored internally). When an agent process crashes, FreePart needs to restore the states. We analyze the framework APIs and identified 1,841 stateful APIs across four frameworks including OpenCV, Caffe, PyTorch, and TensorFlow. We further analyze them and identified that they are APIs for initialization, visualization (*i.e.*, GUI), and data processing.

- **Initialization.** These are APIs that are used during the initialization of the program (*e.g.*, `cv::setNumThreads`). We



(a) Original Program

(b) Partitioned Program (4 partitioned functions). The loop (for at line 2) affects the partitioning of sub functions (e.g., `show()`'s infinite loop).

Figure 17: Challenges in Application-based Partitioning.

observe that they set the state once and do not change it, meaning that we do not need to store the states. After restarting the agent process, simply re-executing the initialization code will restore the state.

- **GUI APIs.** GUI APIs maintain the states for GUI components, while those can be restored by running them again. For example, after a visualization agent process crashed, executing `imshow()` again without restoring the GUI states does not cause issues.
- **Data processing APIs.** Data processing APIs maintain the states internally, and FreePart has to store and restore them, otherwise, they would lead to an incorrect execution. For example, `tf.estimator.DNNClassifier.train` maintains the current state of the training data of a model. For them, we store their states periodically. We find 1,056 APIs in this category.

A.3 API Coverage

Table 11 gives the coverage of dynamic analysis on different APIs.

Framework	API Coverage	Code Coverage
OpenCV	80.4% (424/527)	91%
PyTorch	82.8% (111/134)	84%
Caffe	91.9% (103/112)	76%
TensorFlow	82.6% (2,236/2,704)	73%

Table 11: Coverages of Dynamic Analysis for API Categorization.

Application	Lazy Data Copy	Non Lazy Data Copy
Face_classification	18,722	2,993
FaceTracker	252,892	5,987
Face_Recognition	22,116	8,964
lbpcascade_animeface	1,910	342
EyeLike	29,638	995
Video-to-ascii	6,788	1,997
OMRChecker	7,914	2,357
Libfacedetection	109,258	3,970
EmoRecon	43,698	2,981
OpenPose	369,822	11,958
MTCNN	1,818	202
SiamMask	20,650	111
CycleGAN-and-pix2pix	25,382	2,985
FAIESEQ	21,818	2,992
PytorchGAN	136,602	29,733
YOLO-V3	8,678	333
StarGAN	60,482	2,983
EfficientNet-Pytorch	2,668	168
Semantic-Segmentation	11,278	276
DCGAN-TensorFlow	5,255	107
See in the Dark	6,832	149
CapsNet	4,252	78
Style-Transfer	2,187	128
Total	**1,170,660 (95.08%)**	**82,789 (4.92%)**

Table 12: Statistics of Lazy Data Copy Operations.

A.4 List of Categorized APIs

In Section 4.2, Table 4 shows a few examples of framework APIs with their categories. We list the full list of APIs on GitHub [30].

A.5 Lazy Data Copy

Table 12 shows the number of lazy data copy and non-lazy data copy operations observed during our evaluation. As shown in the table, the lazy data copy occupies 95.1% of the total data copy operations, meaning that it significantly contributes the performance optimization.

A.6 Additional Discussion

Handling Unauthorized Backward Data Flow. If a target application has a complicated backward data flow that cannot be detected by state-of-the-art static analysis techniques, FreePart may fail to identify them, and functionalities that depend on the undetected backward data flow may break. During our experiments, we do not encounter complicated backward data flow cases that break our static analysis. In addition, our empirical study on 56 popular applications (mentioned in **Generality** paragraph) shows that applications typically follow the unidirectional data flow pattern.

Stateful APIs across Agent Processes. A stateful API stores its state during an invocation (on global variables or heap buffers), and the stored state affects the behavior of the API calls in the future (*e.g.*, `strtok` is an example). There are two types of stateful APIs: (1) an API that does *not share the state* with other APIs and (2) an API that *share the state* with other APIs. For the first type, as long as all the API calls are made by the same process, it does not cause any problems. For the second type, if such APIs are executed in different processes, the states will not be shared, breaking the behavior of the target program. To ensure that this would not happen, we run our hybrid analysis to find APIs that access the same memory space (*e.g.*, via arguments, heap buffers, and global variables). To this end, we identify 4,778 and 21 APIs that belong to the data-processing and visualization process, respectively. Among them, we check if there are APIs sharing the states categorized into different processes. We find that debugging/profiling APIs (*e.g.*, tf.debugging.experimental.enable_dump_debug_info()) follow the pattern. Note that such APIs are only reading the profiling data, while other APIs write the data. Our separation essentially separates the profiling data. It does not break the functionality while the profiling data will be collected and reported in each process separately.

Finer-grained System Call Restriction. Observe that our per-agent process system call restriction might be coarse-grained if there are many framework APIs belonging to an agent process. For example, Fig. 12-(c) shows that 9 system calls are permitted in the data loading agent process. In this case, if a vulnerability in `CascadeClassifier::load()` is exploited, the attacker can also access the `ioctl` system call while the system call is *not* required for the framework API. If the restriction is applied per framework API, the attacker has to exploit one of the two methods in `VideoCapture` to access the `ioctl`.

However, implementing the per API system call restriction is not trivial with `seccomp`. Specifically, FreePart uses `PR_SET_NO_NEW_PRIVS`, meaning that only one allowed system call list can be applied for a process. Hence, to support n framework APIs with different required system calls, FreePart needs to run n different processes. For example, Fig. 12-(b) shows three framework APIs that require different sets of system calls. To apply per framework API system call restriction, we need to have three separate agent processes for each API. Unfortunately, running each framework API in a separate process often causes a number of additional IPCs, leading to significant overhead. For example, two data loading APIs in PyTorch, *i.e.*, `datasets.MNIST` and `torch.utils.data.DataLoader`, are operating on the same data. In particular, `torch.utils.data.DataLoader()` takes data returned by `datasets.MNIST()`[11]. If the two APIs are executed in separate processes, there will be an additional IPC between the two processes. In Fig. 12-(b), the two methods of `VideoCapture` share various data, meaning that running them in separated processes would substantially increase the number of IPCs due to the shared data.

Note that while FreePart applies the system call restriction for each agent process, not for each framework API, it does not mean that FreePart cannot further partition each agent process. FreePart supports multiple sub-partitioned agent processes (*e.g.*, 3 agent processes where each of the processes runs a single framework API in Fig. 12-(b)). An alternative can be assigning multiple closely related framework APIs to a single agent process. Intuitively, methods of the same class share more data than methods of different classes. Hence, instead of having 3 data loading agent processes for Fig. 12, we can have 2 data loading agent processes: the first process for `CascadeClassifier::load()` and the second process for the two methods of `VideoCapture`. This would minimize the overhead caused by additional IPCs while providing better security. However, further partitioning the agent process requires manual effort. To this end, FreePart allows the execution of sub-partitioned agent processes if one manually partitions it.

Framework/Program Updates. When a target program is updated, a user needs to reapply FreePart on the updated target program and framework. If the framework is not updated and the target program does not use new framework APIs, the reapplying process simply inserts a few lines of code to hook data objects and functions, which does not cause compatibility issues. If a framework is updated, it should be analyzed again.

Impact on Timing. FreePart changes timings of execution since the target program needs to communicate with isolated processes. However, while it can introduce performance overhead at runtime, FreePart does not introduce logical vulnerabilities such as TOCTOU (Time-of-Check to Time-of-Use) [1, 103]. Note that if a target application already has them, FreePart may affect the chance of the exploitability of the vulnerability due to the performance overhead. In any case, the vulnerability will be contained within one isolated process.

A.7 Additional Case Study: StegoNet Trojan Attack

Liu et al. [52] propose an evasive attack that delivers malicious payload in DNN (Deep Neural Network) models. The attack is stealthy because it uses model parameters as a payload injection channel. We decide to use this attack to show FreePart's effectiveness, particularly against the new stealthy and evasive attacks. We use PyTorch [76] to reproduce the attack. Specifically, we create a malicious DNN model containing malicious code. The model is

[11] `r=datasets.MNIST(...); torch.utils.data.DataLoader(r,...)`

loaded by `torch.load()`. Note that loading a DNN model belongs to a Data Processing process because data processing APIs depend on the loaded model.

Mitigation. Since the model is loaded and executed in the Data Processing process, the malicious payload is contained. The original StegoNet paper uses a fork bomb as an example malicious payload. Our analysis shows that none of the data-processing APIs in the frameworks we support requires `fork()`, meaning that FreePart restricts the use of `fork` system call. The attack is successfully prevented.

Mitigation on Other Applications using PyTorch. We also pick two real-world programs to understand the effectiveness of FreePart in mitigating StegoNet attacks. Specifically, we pick a program that analyzes a medical image (*i.e.*, CT image [109]) and another program that does OCR (Optical Character Recognition) on the tax invoices [35].

The first program contains several sensitive pieces of information: the patient's CT image, name, age, and phone number. With FreePart, the patient's name, age, and phone number exist in the target process. The patient's CT image exists in the data loading process. However, the data loading process does not store previously loaded CT images, meaning that it does not contain other users' CT images. Exploitations can happen in both data loading and data Processing processes. However, as discussed above, sensitive data are not accessible.

The second program contains tax images and personal information extracted from the image, such as an address, taxpayer ID, and bank account number. Similar to the first program, input images exist in the data Loading process. However, it only has the currently processed image. Moreover, all other sensitive data exist in the target program process, which is not accessible from the data loading and data processing processes, where exploitations can happen.