BRAID: Uncovering Malware Family Relationships in Data-Scarce Settings

Kevin Valakuzhy[1](✉), Miuyin Yong Wong[2], Omar Alrawi[1], Angelos D. Keromytis[1], Manos Antonakakis[1], and Fabian Monrose[1]

[1] Georgia Institute of Technology, Atlanta, Georgia, USA
{kevinv, alrawi, angelos, manos, fmonrose}@gatech.edu
[2] University of Maryland, College Park, Maryland, USA
miuyin@umd.edu

Abstract. Malware analysis often occurs in low-support settings, where only a few labeled samples are available and analysts require rapid feedback to guide investigation. In these conditions, existing malware family classifiers—largely driven by learning-based techniques—offer limited assistance. We present `BRAID` (Behavioral Relationship Analysis via Implementation Details), which exploits the observation that while malware families may share similar high-level behaviors, the code that implements them is often family-specific. By deriving implementation-level behavioral abstractions, this approach links related samples and separates them from superficially similar but unrelated malware using minimal data. To balance coverage and scalability, it selectively disassembles only behavior-relevant code across multiple memory snapshots, preserving visibility into multi-stage malware while reducing disassembly time by over an order of magnitude. Evaluated on diverse public datasets including up to 153 families with one labeled sample per family, our approach improves macro F1 by more than 2x on supported samples relative to the state-of-the-art. The uncovered connections expose undocumented cross-family code sharing and family-specific traits useful for threat intelligence.

Keywords: Malware Family Classification · Malware Behaviors

1 Introduction

Malicious software remains a costly and persistent threat to modern computing systems. The financial burden of responding to malware-driven breaches routinely exceeds $4 million per incident on average, even before accounting for long-term remediation and business disruption [30]. In 2024 alone, a single ransomware attack caused an estimated $2.3 billion in losses [1] and exposed sensitive information of more than 190 million individuals [2]. With malware-driven cybercrime damages projected in the trillions of dollars annually, there is an urgent need for effective analysis of emerging threats.

Limiting the impact of novel malware depends critically on timely manual analysis. While automated systems handle commodity malware, understanding

new or evolving threats still relies heavily on human experts. Ugarte-Pedrero et al. [36] showed that even with extensive automation, a commercial security operation would still require roughly 250 full-time analysts to investigate the daily influx of malware samples. Beyond managing volume, analysts must rapidly decide which samples warrant deep inspection, as remediation is most effective when performed within hours of first observation [39].

To manage volume and time pressure, analysts use prior knowledge of malware families to accelerate the analysis of new samples [39]. Family-level classification is a cornerstone of this process, enabling the transfer of insights about capabilities, development lineage, and defensive responses across related samples [6]. However, this strategy breaks down precisely when it is most needed — during the early emergence of a malware family — because existing family classifiers typically require a substantial number of labeled examples per family.

To better assist analysts when evidence is limited, the research community has relied on large, labeled datasets such as MOTIF [16] and BODMAS [44] to train malware family classifiers. While these datasets span hundreds of families, they exhibit a pronounced long-tail distribution, leaving few families with enough samples to support robust training in real-world settings [11]. Crucially, this long-tail regime mirrors the actual conditions analysts face with emerging malware families. In such settings, classifiers frequently conflate samples that reuse common third-party components, such as packers [3], or share standard libraries [22]. Reliable family identification under these constraints requires signals of relatedness that remain stable despite limited training data.

Our key insight is that the implementation of malicious behavior provides a natural and underutilized signal for reliably identifying malware families in data-scarce settings. While different families may share high-level behaviors, the code that implements those behaviors—the family's core logic—often remains distinct. We exploit this observation by deriving implementation-based behavioral abstractions and applying existing binary code similarity techniques to link related malware even in families with few samples. Focusing on code responsible for malicious behaviors offers more stability than approaches considering all code [20], ignores benign libraries, improves robustness in data-sparse settings, and substantially reduces the cost of disassembly and binary code comparison. These efficiency gains are particularly important for multi-process and multi-stage malware. Furthermore, by tracking relationships between the processes that implementations appear in, we can separate family-specific logic from third-party components reused across families. We instantiate these insights in `BRAID`, which operates exclusively on artifacts already produced by standard sandboxes for Windows malware, integrating directly into existing analyst workflows [39].

Our contributions are as follows:

- **Implementation-based relationship discovery**. We introduce a methodology for identifying relationships between malware families based on shared and divergent behavior implementations. Our approach uncovers previously undocumented cross-family code sharing in multi-stage malware and reveals family-specific traits that can inform robust threat-intelligence signatures.

- **Selective disassembly for scalable extraction**. We propose a novel *selective disassembly* technique that targets only the code implementing specific behaviors. This approach reduces disassembly time by over an order of magnitude while maintaining the fidelity required for binary code similarity.
- **Improved few-shot family classification**. We use implementation-derived behavioral relationships in a hybrid graph-based labeling approach to propagate labels from sparse seeds. The resulting malware clusters align better with expert analyst labels than aggregated AV labels. In the single-labeled-sample setting, this approach improves macro F1 by 2× on supported samples over current state-of-the-art.

2 Background

Evaluating malware family classifiers under realistic conditions requires datasets that span numerous families and reflect operational diversity [11]. Consequently, recent work [11, 17, 21, 42, 45] has converged on a small number of large, public benchmarks covering hundreds of malware families. Two such datasets are MOTIF [16], which contains 3,095 Windows malware samples across 454 families, and BODMAS [44], which includes 57,293 samples across 581 families. MOTIF is curated from five years of threat intelligence reports published by 14 major cybersecurity organizations and remains the largest publicly available Windows malware dataset with family labels directly supported by public reports.

Despite their scale, these datasets exhibit a pronounced long-tail distribution in family size: only a few families contain more than 100 binaries (e.g., just over 9% in BODMAS). This sparsity undermines supervised and few-shot learning approaches, which typically require many examples per class to establish stable decision boundaries. As a result, classification performance is skewed by well-represented families, obscuring behavior in the data-sparse setting. Crucially, this long-tail phenomenon reflects the real-world conditions analysts face when encountering emerging or niche malware, where labeled examples are scarce. In such settings, robust family classification requires signals of relatedness that remain reliable even when statistical aggregation across many samples is infeasible.

A natural candidate for a reliable signal is the implementation of malicious behavior. While malware families often share high-level goals, such as process injection or ransomware deployment, these goals can be achieved with different code. Behavioral taxonomies such as MITRE ATT&CK [33] capture what malware does but not how it is implemented. It is precisely this *how* — i.e., the code that realizes malicious logic — that encodes family identity. Focus on these specific regions of code is essential to avoid complications that arise from code reuse. For example, the use of packers and obfuscation frameworks introduces common code across otherwise unrelated families [3]. Shared compiler-generated code and standard libraries further confound naive code-similarity approaches [22].

Our key observation, however, is that the diversity of behavior implementations is not incidental. For instance, although 168 distinct families in MOTIF exhibit behavior matching the signature *Create Process with Hidden Window*, we

observe 465 unique implementations. We found that these variations arise from many sources, including design choices, such as passing data via function arguments versus global variables, and obfuscation techniques, including dynamic API resolution and junk code insertion. Rather than obscuring relationships, this *implementation-level diversity provides the discriminative structure* needed to link samples within a family and separate them from unrelated malware.

3 Related Work

Many prior works classify malware into families using supervised learning techniques [11, 16, 19, 20, 41, 43, 44]. These approaches are effective given sufficient labeled data; however, acquiring high-quality labels at scale is costly. Li et al. [20] mitigate this limitation while addressing concept drift by training on assembly-level Control Flow Graphs (CFGs) of samples taken from periods before and after concept drift occurs to identify stable features to rely on. Their approach, however, fails to hone in on the code for malicious behaviors and does not operate on memory snapshots captured during dynamic analysis—two key criteria we show are important when operating in low support settings.

Alternatively, semi-supervised learning leverages large volumes of unlabeled or weakly labeled samples [21, 31, 42]. Rieck et al. [31], for example, propose an incremental approach that alternates between clustering and classification, deferring the classification of small clusters until enough evidence accumulates. Wu et al. [42] address noisy labeling by identifying likely labeling errors and then applying a semi-supervised learning strategy tailored to imbalanced malware family distributions. Li et al. [21] introduced MalMixer, a system that separates interpolatable and non-interpolatable features, enabling classification of some families with as little as a single labeled sample. Despite diminishing the requirement for labeled samples, semi-supervised approaches still require enough samples, both labeled and unlabeled, to construct statistical models for each family. As a result, they struggle to classify families with few collected samples, including those encountered during early stages of emergence—precisely when timely and accurate classification is required. These conditions lead Li et al. [21] to state that "accurately classifying small families is not the intended application" of their approach. Additionally, the static features that certain approaches [21, 42] rely on are especially vulnerable to static obfuscation (e.g., packing).

Prior work has also explored the use of binary code similarity and runtime behaviors in conjunction with unsupervised learning techniques to identify relationships between malware samples [5, 8, 10, 14, 15, 27]. Cozzi et al. [8] identify relationships among Linux IoT malware, but their approach relies on access to debugging symbols, publicly available source code, and the absence of static obfuscation like packing, assumptions that rarely hold for Windows malware. To bypass packing and other forms of static obfuscation, Mirzaei et al. [27] extract executable code from multiple process-level memory snapshots per malware sample. While effective at increasing code coverage, these techniques do not account for performance costs (easily on the order of tens of minutes) incurred when dis-

assembling and analyzing large numbers of snapshots. Such costs are magnified in operational settings where analysts work in highly iterative workflows [38,39].

At the other extreme, Hu et al. [14, 15] reduce analysis time by capturing a single memory snapshot from the malware's initial process. Although this improves efficiency, it can miss code executed in later stages or in additional processes spawned during execution, which often contain the family-defining malicious logic that analysts seek to understand. Alternative methods, such as those by Dam et al. [10] and Bayer et al. [5], cluster malware based on data dependencies in system calls. These approaches avoid code and memory analysis altogether and therefore scale well, but they overlook cases in which malware behaviors are shaped by third-party obfuscation tools or by multi-stage execution. As a result, they provide limited support for analysts attempting to disentangle family-specific behavior from shared infrastructure code.

Our Work: We present an implementation-level abstraction that works in lower support settings than prior work. Our approach departs from prior work by exploiting execution stage structure: in practice, code from droppers and obfuscation tools executes early, while family-defining logic appears later. This temporal separation distinguishes family-specific implementations from reused infrastructure code, enabling reliable linkage of related samples. Our design also accounts for the realities of human-driven malware analysis, where efficiency is essential to avoid disrupting analysts' tightly coupled loop of experimentation.

4 Methodology

Figure 1 provides an overview of our multi-phase workflow for extracting and comparing implementations of malicious behaviors. In Phase ①, we capture runtime memory snapshots of processes created or injected into during malware execution. In Phase ②, we identify API calls associated with specific behaviors, which we refer to as behavioral anchors. Using these anchors, Phase ③ performs selective disassembly to extract the corresponding implementations, defined as the binary functions containing each anchor. Finally, in Phase ④, we compare extracted implementations against a hierarchically organized datastore of previously observed implementations to identify relationships between implementations and their associated malware samples.

The emphasis on efficiency throughout our design is due to the realities of human-driven malware analysis. An observational study of security professionals by Votipka et al. [38] shows that analysts reason about novel malware through an iterative process, where the impact of delays is magnified. Accordingly, we avoid adding burdens to malware analysis pipelines [6,39] by using behavioral artifacts already produced by standard sandbox environments and ensuring that our analysis time remains negligible relative to sandbox runtime. Aligning scalability and accuracy with analyst workflows helps our design support timely decision-making when automated methods are least effective due to limited data.

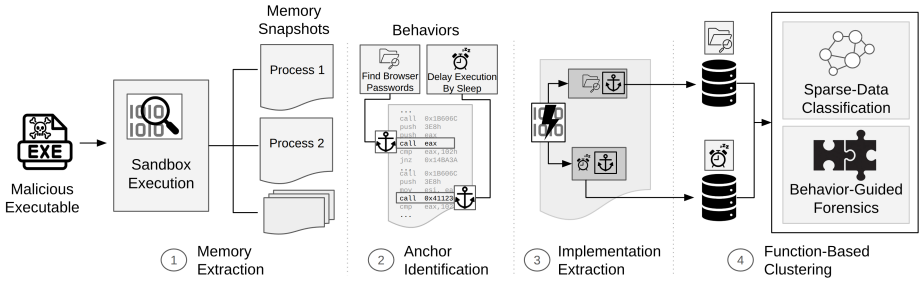

Fig. 1: We extract memory snapshots and identify anchors (⚓) for different behaviors (🗁, ⏰), extracting the surrounding functions using our selective disassembler (🗲). These functions provide interpretable connections between malware samples that can aid analysts in analyzing novel malware.

4.1 Phase ①: Runtime Memory Extraction

To capture implementations of the malware's behaviors, it is critical that the malware executes its malicious logic in the analysis environment. This is essential for recovering implementations hidden by static obfuscation techniques, such as packing, which conceal code until runtime. However, malware often suppresses malicious behavior if it detects that it is being observed [24] and can evade analysis by injecting code into benign processes, distancing execution from the original malicious executable. Although analysis environments are constantly improving, malware also continue to update their evasive methods. For our approach, we expose malicious behaviors using a malware sandbox — the primary dynamic analysis tool relied on by analysts in practice [6, 39].

Once behaviors are executed, their binary implementations can be captured via memory snapshots. To bypass obfuscation techniques such as packing, snapshots are taken at process termination [23]. To reduce disassembly overhead, we restrict snapshots to memory regions that were executed or written to by the original malware process or its descendants, accounting for techniques such as process injection. Nevertheless, disassembling these snapshots remains time-consuming; as shown in Table 1, disassembly requires approximately five minutes per sample on average, and sometimes exceeds 30 minutes. Consequently, minimizing disassembly overhead is a central consideration in our approach, motivating the targeted analysis techniques introduced in subsequent phases.

Some advanced obfuscation tools re-obfuscate code after execution, removing behavior implementations from post-execution memory snapshots. While we considered addressing these cases by including five additional snapshot triggers (i.e., on process start, first network activity, and when newly written memory regions are executed, modified, or are made executable), these triggers increased the number of extracted implementations by less than 1%. This result aligns with previous measurements on the rarity of re-obfuscating packers [35]. Given the limited benefit, our prototype relies solely on post-execution snapshots.

Table 1: Reductions in average disassembled data and analysis time, relative to the baseline approach (A_1). Analysis conducted on 100 randomly selected MOTIF binaries.

ID	Data (KB)	Time (s)	Data Red.	Time Red.
A_1	8,900	341	–	–
A_2	3,100	85	2.9×	4.0×
A_3	66	1	135×	341×

A_1**:** Final snapshot per memory region (IDA Pro). A_2**:** A_1 + behavior-anchored snapshots only (IDA Pro). A_3**:** A_2 + behavior-anchored selective disassembly.

4.2 Phase ②: Identifying Behavioral Anchors

Next, we search the memory snapshots to locate the binary code responsible for the malware's executed behaviors. Intuitively, many behaviors can be located by the key pieces of code, which we coin *behavioral anchors*, required to implement the behavior. Some types of effective anchors are system calls and Windows APIs, as invoking these is required to read or modify resources managed by the operating system.[1] To find API calls that can serve as anchors, we use behavior signatures already present in the malware sandbox. While constructing new signatures is outside the scope of this work, prior work by Comparetti et al. [7] notes the practicality of manually adding behavior signatures, citing the rich context and high level of abstraction offered by dynamic analysis.

After identifying APIs that match behavior signatures, we determine their locations (i.e., virtual addresses) within the previously taken memory snapshots via the function call stack [23]. We then validate each anchor by confirming that it points to bytes in memory that disassemble to an instruction capable of invoking the API, either a call or jmp instruction. These validated behavioral anchors pinpoint the binary-level functions we ultimately use as our behavior implementations and allow us to discard snapshots that lack anchors. This filtering reduces disassembly time, as shown in Table 1, reducing the average disassembly time for each sample's memory snapshots to 85 seconds (A_2). In the next phase, we further reduce this time with a technique we have coined *selective disassembly*.

4.3 Phase ③: Extracting Implementation-Specific Artifacts

For our approach, we choose to extract behavior implementations at the function level, as functions provide a natural unit for grouping code [23] and enable the use of established function-level techniques to compute similarity. Specifically, we consider any function that contains a behavioral anchor as a behavior implementation. While conventional disassembly tools like IDA Pro and Ghidra can recover these implementations from memory snapshots, most of their runtime is spent processing code in unrelated functions. In practice, we observe that behavior implementations account for less than 1% of the bytes of code in a snapshot.

[1] Special instructions, like RDTSC or CPUID, can also serve as behavioral anchors.

This imbalance motivates our selective disassembly approach, which targets only the functions that implement observed behaviors.

Algorithm 1: Selective Disassembly

```
1  Function SelectiveDisassembly(anchor_addr, δ):
2      RecursiveDescent(anchor_addr);
3      end_addr = anchor_addr;
4      start_addr = GetPriorAPICallAddress(anchor_addr);
5      while NumInstructionsDisassembled() < δ do
6          DisassembleRange(start_addr, end_addr);
7          end_addr = start_addr;
8          start_addr = GetPriorAPICallAddress(start_addr);
9      end while
10     implementation = GetFunction(anchor_addr);
11     return implementation;
```

Our selective disassembly approach, shown in Algorithm 1, begins at the address of the behavioral anchor ($anchor\_addr$), a known instruction boundary within the target function. We first apply standard recursive descent disassembly [28] (Line 2), which follows control flow to disassemble instructions after the behavioral anchor that belong to the same function. However, recovering instructions that *precede* the anchor is more challenging for variable-length x86 instructions. To address this challenge, we use API call sites collected during execution as additional known instruction boundaries. Specifically, we select the nearest preceding call site to our previous starting point ($GetPriorAPICallAddress$) and disassemble the region between the two points (Line 6).

When the preceding API call belongs to a separate function, recursive descent alone will not cross function boundaries to disassemble the rest of the target function. Therefore, we include a linear sweep approach [28], which assumes instructions are laid out successively, to disassemble adjacent functions after recursive descent completes. We select APIs as starting points progressively farther from the behavioral anchor until we disassemble a configurable threshold δ of instructions (Line 5). The impact of δ is measured in Section 5.1. Finally, in $GetFunction$, we follow control flow to find instructions in the same function as the behavioral anchor. With this approach, we reduce the disassembly time in Table 1 from 85 seconds to under one second per sample (A_3).

4.4 Phase ④: Function-Based Clustering

After extracting implementations of observed behaviors, we model the relationships between malware samples and behavior implementations as a heterogeneous graph to provide the structural foundation for identifying malware families. Formally, we define a graph $G = (V, E)$, where the node set $V = V_S \cup V_I$

consists of memory snapshot nodes V_S, produced during dynamic execution, and behavioral implementation nodes V_I, extracted from those snapshots.

As a preprocessing step, we discard implementations consisting of fewer than five instructions. Such short functions are excluded in binary similarity analyses [23] because they typically correspond to trivial wrappers, such as thin stubs that immediately transfer control to API calls without modifying inputs or outputs. These implementations exhibit little structural diversity and provide limited discriminatory value for distinguishing malware families.

We define three classes of edges to encode relationships among the remaining nodes. First, *extraction edges* link memory snapshots and their extracted behavior implementations. Second, *execution-flow edges* link memory snapshots of related processes, capturing parent–child relationships and code-injection events. Finally, *similarity edges* link behavior implementations that exhibit the same behavior and similar code. More formally, a similarity edge is added between two implementation nodes I_a and I_b that match a common behavior signature if their similarity satisfies $\text{sim}(I_a, I_b) \geq \sigma$, where σ is a similarity threshold. Restricting similarity comparisons to implementations of the same behavior ensures a degree of semantic similarity while substantially reducing the number of required function comparisons. The appropriate value of σ depends on both the similarity metric and the analyst's tolerance for false positives; rather than fixing it *a priori*, we derive suitable values empirically in our evaluation.

To compute similarity between behavior implementations, we represent each function as a string formed by concatenating the opcode bytes of its instructions in sequential order. Using opcode bytes provides a simple normalization that improves robustness to irrelevant binary differences, such as variations in memory addresses [14, 15]. Similarity between opcode strings is measured using *Normalized Levenshtein Similarity* (NLS), which is computed by dividing their Levenshtein edit distance by the length of the longer string and subtracted from 1 to yield a score in the range $[0, 1]$. Despite its simplicity, this metric performs well in practice and balances accuracy and computational efficiency.

This graph representation supports both unsupervised clustering and semi-supervised classification for realistic sparse-label conditions. While our clustering uses a standard community detection technique, we leverage a custom label-propagation approach for our semi-supervised setting. Specifically, we treat a set of labeled nodes from the training data as seeds and apply Voronoi graph partitioning [12] to propagate labels to unlabeled nodes. Each unlabeled node is assigned the label of the nearest seed by shortest-path distance in the graph.

To consistently label snapshots associated with the same malware sample, we assign zero weight to execution-flow edges and assign unit weight to all others. This scheme ensures that snapshots of the same sample receive identical labels while preserving meaningful distances between different samples and their behavior implementations. When multiple seed nodes are reachable at the same minimum distance, ties are broken at random. Nodes that are unreachable from any seed are treated as out-of-distribution and are not assigned a family label.

5 Evaluation

We now assess the benefits of our approach. First, we compare our selective disassembly approach to a state-of-the-art disassembler. Second, we evaluate our implementation-based similarity on families with few examples. Lastly, we share instances where our analyses led to insights on cross-family relationships, both confirming and extending results from professional analyst reports.

Implementation Details. Commercial malware sandboxes are regarded by practitioners as a critical tool for analyzing evasive malware [29, 40]. Accordingly, `BRAID` relies on dynamic behavior signatures used by sandboxes to identify malicious behaviors during execution. For our prototype, we use VMRay Analyzer [2], a widely used sandbox in industry, government, and academic research [29], configured with 64-bit Windows 10 virtual machines. Importantly, our approach is not tied to any specific sandbox and can operate with any environment that provides structured behavioral observations and memory snapshots. Based on the finding of Kuchler et al. [18] that unique code execution by malware plateaus early during runtime, we fix a 180 second timeout for all experiments.

Our selective disassembly component is implemented in Python on top of the Capstone framework [3] (v4.0.2). For comparison, we use the industry standard IDA Pro (v7.6) disassembler as a baseline. All experiments are conducted on a Linux server running Ubuntu 20.04 with two 8-core Intel Xeon E5-2450 CPUs at 2.10 GHz and 96 GB of DRAM. While we report absolute timings for reproducibility, the relative performance improvements we observe stem from design choices in selective disassembly rather than hardware-specific optimizations.

Data Selection. We adopt the MOTIF dataset for our evaluation as its family labels are sourced from expert-curated threat reports, making it particularly well suited for evaluating family-level relationships. We also include the popular BODMAS dataset [44] whose labels are mostly derived through automated antivirus label aggregation, with around 1% assigned by expert analysts.

Table 2: Final datasets used in our evaluation after preprocessing and filtering.

		MOTIF		**BODMAS**	
ID	**Step**	**Samples**	**Families**	**Samples**	**Families**
S_1	Supported PE EXE	2,132	346	39,582	468
S_2	Set canonical labels	2,132	340	39,582	425
S_3	Remove generic labels	2,131	339	32,768	362
S_4	Extract implementations	1,643	277	18,034	253

From these datasets, we assess how well `BRAID` can identify malware families with as few as two samples; Table 2 depicts how we obtain our evaluation

[2] https://www.vmray.com/vmray-analyzer/
[3] https://www.capstone-engine.org/

dataset. Our current prototype is designed to extract assembly code from Windows EXE files, setting aside .NET and DLL files (S_1). To reduce noise from AV labels, we apply ClarAVy [17] to collapse family aliases into canonical names (S_2) and remove family labels corresponding to generic AV detections (e.g., "generic" or "agent") (S_3). These steps substantially impact the AV-labeled BODMAS dataset, but have minimal effect on MOTIF, which addressed these issues during curation. Finally, we retain only samples for which `BRAID` identifies at least one behavior implementation (S_4), yielding our evaluation datasets. We discuss reasons, and potential fixes, for missing behavior implementations in Section 6.

5.1 Benefits of Selective Disassembly

To evaluate the accuracy and efficiency of selective disassembly, we compare it against the state-of-the-art IDA Pro disassembler. We use IDA Pro's output as a baseline for correctness and as a performance benchmark for robust, non-selective disassembly. We evaluate both approaches on 1,000 samples from the BODMAS dataset, rerunning our approach with multiple choices for the bound on the maximum number of instructions disassembled per implementation δ.

Table 3: Impact of δ on selective disassembly. Results are averaged over 1000 samples from BODMAS. Similarity to IDA Pro is computed using NLS.

Approach	δ	**Extraneous Disassembly**	**Time (s)**	**Similarity to IDA Pro**
IDA Pro	N/A	>2,410 KiB	131.1	1.00
Selective Disassembly	200	21 KiB	0.4	0.90
	500	34 KiB	0.6	0.92
	1,000	**40 KiB**	**0.9**	**0.94**
	2,000	61 KiB	1.1	0.94

The results in Table 3 show that selective disassembly recovers functions that are 94% similar to IDA Pro, while requiring only 1% of the disassembly time. Increasing δ beyond 1,000 instructions yields diminishing returns, as only 1% of the functions recovered by IDA Pro exceed this length. Based on this observation, we set $\delta = 1{,}000$ for all subsequent experiments. Comparison of the functions produced by these two disassemblers reveals that differences arise from misinterpreting data between functions as code, a well-known challenge for linear disassembly of Windows binaries [28]. We leave the incorporation of additional disassembly heuristics, such as function start detection [28], to future work.

5.2 Malware Family Identification

To evaluate `BRAID`, we first set the code similarity threshold σ using the MOTIF dataset. Afterwards, we use the BODMAS dataset to compare our malware

family clusters to aggregated AV labels and test our label-propagation approach (see Section 4.4) against state-of-the-art malware family classification.

Threshold selection. We select σ via grid search, choosing the value that yields malware clusters on MOTIF that most closely match expert-provided family labels. For each value of σ, we construct a graph using `BRAID` and use the Louvain community detection algorithm [4] for clustering. Cluster quality is assessed using Adjusted Mutual Information (AMI) [37], which measures the agreement between two clusterings while correcting for chance. We also report AMI scores obtained on the BODMAS dataset to assess cross-dataset generalization.

Table 4: AMI for different values for σ. Scores better than AVClass2 are bolded.

σ	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
MOTIF	0.425	0.501	0.644	**0.733**	**0.755**	**0.762**	**0.764**	**0.753**	**0.742**	**0.720**
BODMAS	0.699	0.733	**0.794**	**0.803**	**0.807**	**0.807**	**0.801**	**0.797**	**0.792**	**0.786**

As shown in Table 4, $\sigma = 0.7$ produces clusters that align most closely with the MOTIF ground truth. Notably, this threshold also achieves near-peak performance on BODMAS, indicating that the selected value generalizes beyond the dataset used for tuning. For additional context, we compare `BRAID`'s clustering results against those produced using aggregated antivirus labels generated by AVClass2 [32]. `BRAID` achieves a 9% higher AMI than AVClass2 (0.76 vs. 0.70), highlighting the value of the relationships captured by our approach.

Classification Results. With σ fixed, we evaluate family classification performance with respect to (i) the number of families and (ii) the number of labeled training samples. We use the approach of Li et al. [20] as a baseline due to its reported strong performance relative to other family classification models, publicly available implementation, and ability to handle changes in malware over time (i.e., concept drift) with as few as ten samples. Their approach achieves these outcomes by training on data from pre- and post-drift to select features that remain reliable as malware changes. To simulate concept drift, we use collection timestamps from the BODMAS dataset to select, for each family, training samples that were collected before testing samples. For Li et al.'s approach, we use the earliest 50% of training data as pre-drift and the remainder as post-drift. When only one training sample is available for a family, we include it in both training sets. We remove any samples for which Li et al.'s data collection pipeline fails to extract features to ensure a fair comparison under full data availability.

The top half of Figure 2 shows the performance of each approach with one training sample per family. We add families from largest to smallest, mimicking family size thresholds from prior work [21], up to all 153 families with at least 2 samples to ensure presence in both training and testing data. We report macro statistics to weight each malware family equally despite the long-tail distribution. As expected, performance degrades as the number of families increases, but

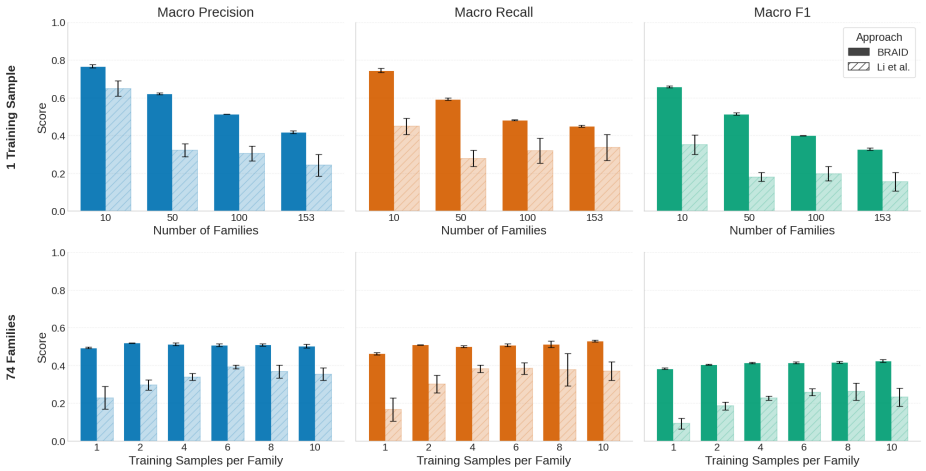

Fig. 2: Comparison of BRAID and Li et al. [20] across varying numbers of families and training samples. Error bars indicate standard deviation across three runs.

BRAID maintains a steady advantage, achieving an 88% improvement in Macro F1 over the baseline with 10 families and a 111% improvement with 153 families. Additionally, our approach exhibits lower variance across runs.

Our second evaluation examines the impact of increasing the number of labeled samples. We select the 74 families from BODMAS with at least 11 samples, using the first 10 for training and the remainder for testing. Holding the test set fixed, we increase the number of training samples per family from 1 to 10, adding samples in temporal order. The bottom half of Figure 2 shows that our approach remains stable, whereas Li et al.'s approach [20] markedly improves as training samples per family increase (up to 6). Upon closer inspection, we find that sensitivity to training set size is heavily dataset dependent. Repeating this experiment on MOTIF resulted in even greater performance gains as the number of training samples increased, with further details in Appendix A.1.

5.3 Overlap with Benign Executables

BRAID is designed to cluster known malware families, a practical focus given the availability of high-performance malware detectors [20]. However, some behaviors (e.g., remote access) are not inherently malicious, and their signatures may occasionally match benign software. To evaluate this effect, we analyzed 1,000 benign Windows binaries from the Dike dataset[4], which is used to evaluate malware detection systems [13]. Although BRAID extracted 40 implementations, the maximum NLS similarity between benign and malicious samples was only 0.21. This is well below the thresholds (σ) used in our evaluations, suggesting that BRAID primarily captures connections specific to malware.

[4] https://github.com/iosifache/DikeDataset

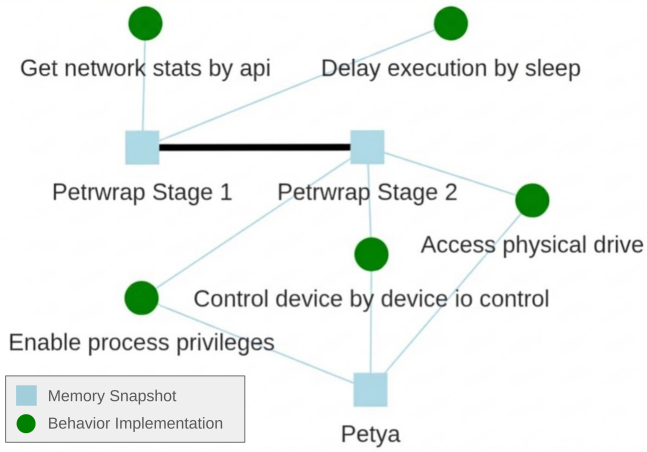

Fig. 3: Unique behavior implementations found in *petrwrap* and *petya*. Notice shared implementations between *petya* and the 2nd stage of *petrwrap*.

5.4 Forensic Analysis using Behavioral Abstractions

We further demonstrate the value of `BRAID` by using behavior implementations to explain inter-family relationships in the MOTIF dataset. Without understanding the cause of cross-family code sharing, novel malware samples may be incorrectly grouped with known families, delaying remediation efforts [36]. To study such relationships, we identify 21 clusters that contain samples from exactly two malware families and randomly select 25% (5 clusters) for inspection. In four cases, the cross-family code similarity arises because malware from one family functions as a dropper that executes malware from another family as a payload, a relationship that is difficult to uncover without identifying the different stages of execution. The final case, *warzone* and *avemaria*, reflects an alias relationship missing from ClarAVy's list used to preprocess the data in Section 5.

`BRAID` identifies all four dropper–payload relationships through a common pattern: payloads execute in separate processes with distinct behavior implementations. Accordingly, when a memory snapshot corresponding to a newly spawned or injected process exhibits different behavior implementations than the snapshot of its parent process, we mark the child snapshot as a new stage. Figure 3 illustrates how stage boundaries expose code-sharing relationships that would otherwise be obscured, improving family classification in data-sparse settings while providing actionable explanations for cross-family links.

We further find that execution stages help identify cross-family relationships introduced by shared third-party tools. Figure 4 illustrates one such case, where an implementation is shared between the *locky*, *flokibot*, and *cerber* malware families. Closer inspection reveals that this connection only appears in the first execution stage of these families—a stage in which malware commonly relies on external tooling. Manual analysis confirms this intuition, finding that the function is added by the Nullsoft Scriptable Install System (NSIS), a Windows

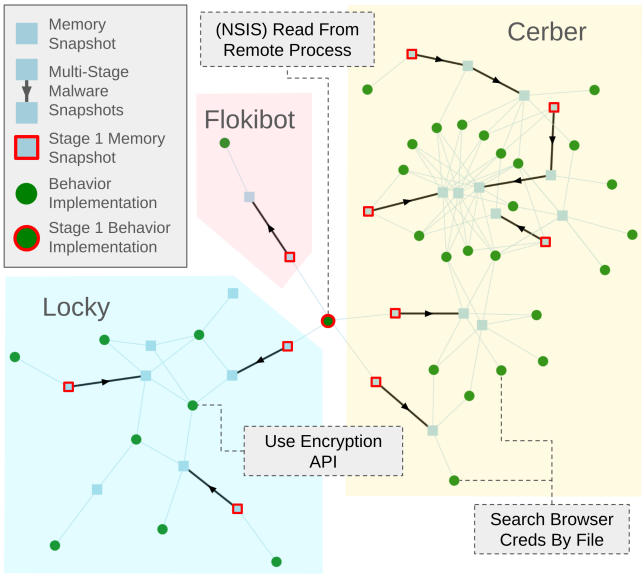

Fig. 4: Implementations from three different families. Different stages of execution help to distinguish family-specific behavior from third-party tooling behavior.

installer framework used to hinder analysis [9]. In contrast, implementations expressed in later stages align with family-specific functionality. For example, the second stage of *locky* samples share an implementation of the *Use Encryption API* behavior, reflecting *locky*'s role as ransomware. Similarly, in *cerber*, the *Search Browser Creds By File* behavior appears only in the second stage of certain samples, a capability that open-source intelligence (OSINT) reports state was added in newer versions.

Unveiling Undocumented Family-Specific Code. Behavior implementations that are unique to a malware family provide a powerful basis for constructing robust detection signatures. To evaluate how `BRAID` can strengthen family identification, we measure its ability to discover undocumented, family-specific traits. We use OSINT reports from MOTIF as our baseline of public knowledge, selecting ten that explicitly reference hashes for at least two samples from the same family. Two authors independently searched each report for behaviors attributed to each family and assessed whether the report provided the assembly code for these behaviors. The Cohen's kappa [26] intercoder agreement between the two authors was 0.9, indicating strong consistency in their assessments.

Table 5 shows that OSINT reports frequently document high-level behaviors without providing code-level evidence. `BRAID` bridges this gap by providing both behavioral evidence and the family-specific code that realizes it. For example, in a Kaspersky report discussing *azorult* malware, our analysis uncovered 11 implementations of malicious behaviors that were unique to the *azorult* family and

Table 5: Baseline evidence for malware family groupings provided by OSINT reports, and additional behavioral and family-specific functions uncovered by BRAID.

		OSINT Report		BRAID (New Evidence)	
Family	**Intel**	**Behavior**	**Code**	**Behavior**	**Family-Specific Functions**
Azorult	Kaspersky	✓	✗	✓	**11**
Buer	G-Data	✓	✗	✓	**2**
Danabot	Proofpoint	✓	✓	✗	0
Deloader	Fortinet	✓	✓	✓	**1**
Gandcrab	Fortinet	✓	✓	✓	**9**
Locky	MalwareBytes	✗	✗	✓	**4**
Melcoz	Kaspersky	✓	✗	✗	**1**
NeutrinoPOS	Kaspersky	✓	✓	✓	**10**
Trickbot	Bitdefender	✓	✗	✓	**1**
Zebrocy	CISA	✓	✗	✗	0

absent from the original report. Across the ten examined reports, we found undocumented shared behavior implementations in seven families, with five reports lacking code evidence entirely. Overall, our approach can surface family-defining traits missing from OSINT, adding clarity to malware similarities and supporting the construction of more resilient detection signatures.

6 Limitations

While BRAID works when behavior implementations are extracted, samples without such implementations require complementary techniques. In our prototype, these cases arise when behaviors are not triggered during execution or when relevant memory regions are excluded from process-end snapshots. Both issues can be addressed in future work by expanding behavior coverage through additional signature sources (e.g., [25]), adding anti-evasion techniques [11], tagging memory regions at execution time to ensure capture at termination, and even enabling memory snapshots at runtime when behaviors are observed to ensure the call site is present in memory. To handle advanced, virtualization-based packers, BRAID can be integrated with recent deobfuscation techniques [34].

For unlabeled samples from previously unseen families, three outcomes arise: (i) unique implementations, (ii) shared implementations with existing families, or (iii) behaviors not captured by current signatures. In the first case, samples remain unlabeled after propagation, providing a natural starting point for manual analysis and identification of new families. In the second, shared implementations raise ambiguity between new families and variants. While this distinction ultimately requires human judgment, our approach helps by producing clusters that align closely with analyst-defined groupings (per Table 4). In the third case, we posit that BRAID remains extensible through addition of signatures for new behaviors in MITRE ATT&CK and similar taxonomies.

A threat to our approach is the rise of LLM-generated malware. As shared code-generation tools, LLMs increase the likelihood that unrelated authors produce similar implementations of the same behavior. While model diversity, prompt variation, and sampling randomness introduce some variation, they do not eliminate this convergence. As a result, similarity-based classification may capture common generation patterns rather than true lineage, leading to broader implications on how malware families are defined and interpreted in the future.

7 Conclusion

We provide an end-to-end approach for expeditiously finding connections between malware samples in low-support settings. Through careful choice of methods, we reduce the overhead of extracting implementations of behaviors by over two orders of magnitude, allowing binary code similarity approaches to be integrated into malware analysts' time-sensitive workflows with minimal cost. We also demonstrate how our approach can provide concrete code evidence for assertions in malware reports, explain cross-family code sharing and lineage, and reveal previously undocumented relationships between malware families.

Acknowledgments. This material contains work supported by the Defense Advanced Research Projects Agency under Contract No. HR001123C0035. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Department of Defense (DoD).

A Appendix

A.1 Training Data Sensitivity on MOTIF

For completeness, we repeated the evaluation from Section 5.2 using the MOTIF dataset, identifying 33 families with at least 11 samples that both `BRAID` and Li et al. [20] successfully processed. To avoid overfitting to MOTIF, we set the code similarity threshold σ to 0.6 based on peak performance on the BODMAS dataset (Table 4). Figure 5 shows that the macro statistics from both approaches significantly improve as the number of training samples increases. We surmise that the differing sensitivity to the number of training samples between the BODMAS and MOTIF evaluations is influenced by their collection periods; whereas BODMAS spans only one year, MOTIF covers five years, potentially resulting in greater concept drift.

A.2 Open Science

The resources used in this study are available at https://github.com/kvalakuzhy-gatech/BRAID. This includes the datasets, the references supporting our forensic analysis, and the code for our selective disassembly approach and the overall pipeline presented in Section 4.

Fig. 5: Performance of BRAID vs Li et al. [20] at malware classification on 33 families.

References

1. Unitedhealth group reports second quarter 2024 results. https://www.unitedhealthgroup.com/content/dam/UHG/PDF/investors/2024/UNH-Q2-2024-Release.pdf (2024)
2. Change healthcare cybersecurity incident frequently asked questions. https://www.hhs.gov/hipaa/for-professionals/special-topics/change-healthcare-cybersecurity-incident-frequently-asked-questions/index.html (2025)
3. Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., Kruegel, C.: When malware is packin' heat; limits of machine learning classifiers based on static analysis features. In: Symposium on Network and Distributed System Security (2020)
4. Amira, A., Derhab, A., Karbab, E.B., Nouali, O.: A survey of malware analysis using community detection algorithms. ACM Computing Surveys (2023)
5. Bayer, U., Comparetti, P.M., Hlauschek, C., Kruegel, C., Kirda, E.: Scalable, behavior-based malware clustering. In: Symposium on Network and Distributed System Security (2009)
6. Botacin, M.: What do malware analysts want from academia? a survey on the state-of-the-practice to guide research developments. In: International Symposium on Research in Attacks, Intrusions and Defenses (2024)
7. Comparetti, P.M., Salvaneschi, G., Kirda, E., Kolbitsch, C., Kruegel, C., Zanero, S.: Identifying dormant functionality in malware programs. In: IEEE Symposium on Security & Privacy (2010)
8. Cozzi, E., Vervier, P.A., Dell'Amico, M., Shen, Y., Bilge, L., Balzarotti, D.: The tangled genealogy of iot malware. In: Annual Computer Security Applications Conference (2020)
9. Crofford, C., McKee, D.: Ransomware families use nsis installers to avoid detection, analysis. https://www.mcafee.com/blogs/other-blogs/mcafee-labs/ransomware-families-use-nsis-installers-to-avoid-detection-analysis/ (2017)
10. Dam, K.H.T., Given-Wilson, T., Legay, A.: Unsupervised behavioural mining and clustering for malware family identification. In: Symposium on Applied Computing (2021)
11. Dambra, S., Han, Y., Aonzo, S., Kotzias, P., Vitale, A., Caballero, J., Balzarotti, D., Bilge, L.: Decoding the secrets of machine learning in malware classification: A deep dive into datasets, feature extraction, and model performance. In: ACM Conference on Computer and Communications Security (2023)

12. Erwig, M.: The graph voronoi diagram with applications. Networks: An International Journal (2000)
13. Gao, Y., Hasegawa, H., Yamaguchi, Y., Shimada, H.: Malware detection using attributed cfg generated by pre-trained language model with graph isomorphism network. In: Annual Computers, Software, and Applications Conference (2022)
14. Hu, X., Bhatkar, S., Griffin, K., Shin, K.G.: MutantX-s: Scalable malware clustering based on static features. In: USENIX Annual Technical Conference (2013)
15. Hu, X., Shin, K.G.: DUET: integration of dynamic and static analyses for malware clustering with cluster ensembles. In: Proceedings of the 29th Annual Computer Security Applications Conference (2013)
16. Joyce, R.J., Amlani, D., Nicholas, C., Raff, E.: Motif: A malware reference dataset with ground truth family labels. Computers & Security (2023)
17. Joyce, R.J., Everett, D., Fuchs, M., Raff, E., Holt, J.: Claravy: A tool for scalable and accurate malware family labeling. In: Companion Proceedings of the ACM on Web Conference (2025)
18. Küchler, A., Mantovani, A., Han, Y., Bilge, L., Balzarotti, D.: Does every second count? Time-based evolution of malware behavior in sandboxes. In: Symposium on Network and Distributed System Security (2021)
19. Kurlandski, L., Berger, H., Pan, Y., Wright, M.: Beyond raw bytes: Towards large malware language models. In: Symposium on Network and Distributed System Security (2026)
20. Li, A.S., Iyengar, A., Kundu, A., Bertino, E.: Revisiting concept drift in windows malware detection: Adaptation to real drifted malware with minimal samples. In: Symposium on Network and Distributed System Security (2025)
21. Li, J., Zhang, Y., Huang, Y., Leach, K.: Malmixer: Few-shot malware classification with retrieval-augmented semi-supervised learning. In: European Symposium on Security and Privacy (2025)
22. Li, Y., Jang, J., Hu, X., Ou, X.: Android malware clustering through malicious payload mining. In: International symposium on research in attacks, intrusions, and defenses (2017)
23. Lindorfer, M., Di Federico, A., Maggi, F., Comparetti, P.M., Zanero, S.: Lines of malicious code: Insights into the malicious software industry. In: Annual Computer Security Applications Conference (2012)
24. Maffia, L., Nisi, D., Kotzias, P., Lagorio, G., Aonzo, S., Balzarotti, D.: Longitudinal study of the prevalence of malware evasive techniques (2021)
25. Mandiant: CAPA: Automatically identify malware capabilities. https://www.mandiant.com/resources/blog/capa-automatically-identify-malware-capabilities
26. McHugh, M.L.: Interrater reliability: the kappa statistic. Biochemia medica (2012)
27. Mirzaei, O., Vasilenko, R., Kirda, E., Lu, L., Kharraz, A.: Scrutinizer: Detecting code reuse in malware via decompilation and machine learning. In: Detection of Intrusions, Malware and Vulnerability Assessment (2021)
28. Pang, C., Yu, R., Chen, Y., Koskinen, E., Portokalidis, G., Mao, B., Xu, J.: Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In: IEEE symposium on security and privacy (2021)
29. Pirch, L., Warnecke, A., Wressnegger, C., Rieck, K.: Tagvet: Vetting malware tags using explainable machine learning. In: European Workshop on Systems Security (2021)
30. Ponemon Institute and IBM Security: Cost of a data breach report 2024 (2024), https://www.ibm.com/reports/data-breach
31. Rieck, K., Trinius, P., Willems, C., Holz, T.: Automatic analysis of malware behavior using machine learning. Journal of Computer Security (2011)

32. Sebastián, S., Caballero, J.: Avclass2: Massive malware tag extraction from av labels. In: Annual Computer Security Applications Conference (2020)
33. Strom, B.E., Applebaum, A., Miller, D.P., Nickels, K.C., Pennington, A.G., Thomas, C.B.: Mitre att&ck: Design and philosophy (2018)
34. Sudhir, A., Basque, Z.L., Gibbs, W., Bajaj, A.P., Singaria, P.S., Zakocs, M., Hu, J., Schloegel, M., Bao, T., Doupe, A., et al.: Pushan: Trace-free deobfuscation of virtualization-obfuscated binaries. arXiv preprint arXiv:2603.18355 (2026)
35. Ugarte-Pedrero, X., Balzarotti, D., Santos, I., Bringas, P.G.: Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In: 2015 IEEE Symposium on Security and Privacy. IEEE (2015)
36. Ugarte-Pedrero, X., Graziano, M., Balzarotti, D.: A close look at a daily dataset of malware samples. ACM Transactions on Privacy and Security (TOPS) (2019)
37. Vinh, N.X., Epps, J., Bailey, J.: Information theoretic measures for clusterings comparison: is a correction for chance necessary? In: International Conference on Machine Learning (2009)
38. Votipka, D., Rabin, S., Micinski, K., Foster, J.S., Mazurek, M.L.: An observational investigation of reverse engineers' processes. In: USENIX Security Symposium (2020)
39. Wong, M.Y., Landen, M., Antonakakis, M., Blough, D.M., Redmiles, E.M., Ahamad, M.: An inside look into the practice of malware analysis. In: ACM Conference on Computer and Communications Security (2021)
40. Wong, M.Y., Landen, M., Li, F., Monrose, F., Ahamad, M.: Comparing malware evasion theory with practice: results from interviews with expert analysts. In: Symposium on Usable Privacy and Security (2024)
41. Wu, B., Xu, Y., Zou, F.: Malware classification by learning semantic and structural features of control flow graphs. In: International Conference on Trust, Security and Privacy in Computing and Communications (2021)
42. Wu, X., Guo, W., Yan, J., Coskun, B., Xing, X.: From grim reality to practical solution: Malware classification in real-world noise. In: IEEE Symposium on Security & Privacy (2023)
43. Yan, J., Yan, G., Jin, D.: Classifying malware represented as control flow graphs using deep graph convolutional neural network. In: International conference on dependable systems and networks (2019)
44. Yang, L., Ciptadi, A., Laziuk, I., Ahmadzadeh, A., Wang, G.: Bodmas: An open dataset for learning based temporal analysis of pe malware. In: Deep Learning and Security Workshop (2021)
45. Yang, W., Gao, M., Chen, L., Liu, Z., Ying, L.: Recmal: Rectify the malware family label via hybrid analysis. Computers & Security (2023)