Reducing Malware Analysis Overhead with Coverings

Michael Sandborn*, Zach Stoebner*, Westley Weimer†, Stephanie Forrest‡,
Ryan Dougherty§, Jules White*, Kevin Leach*
*Vanderbilt University, †University of Michigan - Ann Arbor,
‡Arizona State University, §United States Military Academy
{michael.sandborn, zachary.a.stoebner, jules.white, kevin.leach}@vanderbilt.edu, weimerw@umich.edu,
steph@asu.edu, ryan.dougherty@westpoint.edu

Abstract—There is a substantial and growing body of malware samples that evade automated analysis and detection tools. Malware may measure fingerprints ("artifacts") of the underlying analysis tool or environment, and change their behavior when such artifacts are detected. While analysis tools can mitigate artifacts to reduce exposure, such concealment is expensive and limits scalable automated malware analysis. However, not every sample checks for every type of artifact—analysis efficiency can be improved by mitigating only those artifacts most likely to be used by a sample. Using that insight, we propose MIMOSA, a system that identifies a small set of "covering" configurations that collectively and efficiently defeat most malware samples in a corpus. MIMOSA identifies a set of configurations that maximize analysis throughput and detection accuracy while minimizing manual effort, enabling scalable automation for analyzing stealthy malware. We evaluate our approach against a benchmark of 1535 meticulously labeled stealthy malware samples. We further test our approach on an additional set of 1221 stealthy malware samples and successfully analyze nearly 99% of them using only 2 VM backends. MIMOSA provides a practical, tunable method for efficiently deploying malware analysis resources.

Index Terms—Malware analysis, covering sets, artifact mitigation

1 INTRODUCTION

Malware continues to proliferate, significantly eroding user and corporate privacy and trust in computer systems [1], [2], [3], [4]. Malwarebytes 2022 Threat Review reports a 77% increase in malicious software detected from 2020–2021 as well as an 85% increase in detection numbers for Windows business threats compared to 2019 [5]. SonicWall detected around 10 billion malware attacks in 2019 [6]. Keeping abreast of this large volume of malware requires effective scalable malware analysis techniques.

Once a malware sample has been detected and analyzed, automated techniques such as signature matching can quickly identify other copies. Understanding novel malware samples, however, requires lengthy analysis using both automated and manual techniques [7], [8]. Analysts frequently execute samples under laboratory setups [9], [10] using virtualization. This includes not only virtual machine monitors like VMWare [11], Xen [12], and VirtualBox [13], but also tools that depend on virtualization such as Ether [14], HyperDbg [15], or Spider [16]. Executing the malware sample in a controlled environment allows the analyst to observe its behavior safely. If malware causes damage, the damage is limited to the virtualized environment, which can be destroyed and restarted to analyze subsequent samples. Virtualization is now a lynchpin of computer security and analysis applications [17], [18], [19], [20], [21].

As these malware analysis methods have matured, malware authors have in turn adopted evasive, or *stealthy*, techniques to avoid or subvert automated analysis [22], [23], [24]. For example, Chen *et al.* [23] reported that 40% of malware samples hide or reduce malicious behavior when run in a VM or with a debugger attached. Stealthy malware techniques include anti-debugging [23], [25], [26], anti-virtualization [27], [28], and anti-emulation [29]. These methods detect a particular feature, or *artifact*, of the analysis environment which allows the malware to determine if it is being analyzed. When an artifact is detected, the malware can avoid executing its malicious payload, thereby hiding its true function from the analyst. Table 1 summarizes common artifacts, derived from Zhang *et al.* [30]. Studying the behavior of stealthy malware requires that the analyst *mitigate* the artifacts by configuring the environment in a way that prevents detection by the malware. Over time, malware authors have discovered a wide diversity of artifact types, which has increased the time required to manually determine the best mitigation strategy for each malware sample; however, this process has proven difficult to automate. As malware authors discover new artifacts, analysts must develop new mitigations, raising the cost and complexity of analysis.

Balzarotti *et al.* describe how stealthy malware samples check for evidence of analyses and behave differently when they are present [31]. They classify stealthy malware by running samples under multiple environments and using the differences between those runs, especially in terms of patterns of system call execution, to characterize evasive behavior. For example, if a sample is executed under both VMWare [11] and VirtualBox [13], and the VMWare instance does not exhibit malicious behavior, one can conclude that the sample detects VMWare-specific artifacts (e.g., [32]). Many techniques, from machine learning [33] to symbolic execution and traces [34] to hybrid dynamic analyses [35],

among others, have been proposed to tackle this problem of environment-aware malware—even as new black-hat approaches for stealthy evasion (e.g., [36], [37]) emerge.

This paper presents MIMOSA[1] to address the need for high-throughput, low-overhead automated analysis of stealthy malware. MIMOSA's key insight is that any one malware sample is likely to use a small set of artifacts to detect its analysis environment, and that multiple configurations (i.e., provisioned VM instances) can collectively mitigate a large number of these artifacts for a given malware dataset. We propose using *coverings* from the software testing literature [38] to find a small set of analysis configurations that collectively cover (mitigate) the techniques used by most stealthy malware samples while minimizing the cost of each individual analysis configuration. MIMOSA can be used as part of an automated malware analysis or triage system to help detect and understand new stealthy malicious samples.

We extend the previous state-of-the-art to consider both the *cost* and *coverage* of artifact mitigation strategies. Given the popularity of stealthy malware and the increasing number of anti-stealth techniques, the question is no longer whether or not evasion *should* be mitigated, but *which set* of techniques should be used for a particular sample to minimize the likelihood of analysis detection. Since samples often use combinations of artifacts to evade detection [22], this is not a simple decision. Because each stealth mitigation technique comes with associated costs—development time, deployment time, CPU time, memory and disk utilization, runtime overhead, etc.—compared to a bare-metal or bare-VM setup. These costs are critical because the rate at which new malware is deployed [39] combined with the time and resources required to complete each analysis has led to a situation in which analysis time can be a bottleneck [40].

MIMOSA uses a notion of coverings that describe the mitigations associated with classes of stealthy malware as well as the cost (e.g., runtime overhead, development time, system memory, disk utilization, etc.) to deploy these mitigations. We present an algorithmic approach to deduce a small number of Pareto-optimal configurations that maximize coverage (i.e., the fraction of stealthy samples that can be analyzed successfully) and minimize cost, according to a user-provided cost model.

Once we devise a set of covering configurations, we train a classifier that, given a new malware sample, predicts the best single configuration that is most likely to successfully analyze that input sample. In this way, given a large corpus of unseen malware, we ideally execute these samples on existing configurations predicted by our classifier. In the worst case, new configurations must be developed and the classifier finetuned on a random subset of samples from the new corpus. Then we attempt to execute each sample on its predicted configuration, facilitating efficient and scalable analysis on new malware corpora.

To summarize, the main contributions of this paper are:

- A new algorithm for identifying a low-cost set of artifact covering configurations for malware analysis;
- MIMOSA, a system for selecting and deploying covering combinations of artifact mitigations to maximize analysis throughput and accuracy;
- An empirical study of 1535 labeled stealthy malware samples from the wild, demonstrating that MIMOSA achieves high coverage of stealthy malware and high automated analysis throughput;
- A follow-on study of the Bluepill dataset [41] where MIMOSA successfully analyzes 1207/1221 unseen malware samples.
- Open-source software for scalable analysis of evasive malware. MIMOSA is available at https://github.com/sandbornm/MIMOSA.

1. MIMOSA: Malware Instrumentation with Minimized Overhead for Stealthy Analysis.

2 BACKGROUND

Stealthy Malware We call malware *stealthy* if it actively seeks to detect, disable, or otherwise subvert malware analysis tools. Stealthy malware operates by checking for signatures, or *artifacts*, associated with various analysis tools or techniques. For example, a malware sample may invoke the `isDebuggerPresent` Win32 API call to determine whether a debugger is attached to the process—if a debugger is attached, the sample may conclude that an analyst is instrumenting it and change its behavior accordingly. Briefly, stealthy malware samples use artifacts as heuristics to determine if they are under analysis, and change their behavior to subvert the tool.

Stealthy, evasive malware has been studied extensively [40] and is of increasing concern in industrial settings, with companies such as Minerva and Lastline marketing solutions for detecting stealthy malware. In addition, stealth is often a property gained through the use of packers [42], [43], [44] that can systematically change malware statically to evade detection and subvert analysis. Thus, there is a need for defensive methods that can keep up with the escalating arms race with malware.

An *artifact* is information about the execution environment that a malware sample can use to determine if it is running non-natively. For example, if a malware sample checks whether a debugger is attached to it, that sample may behave differently in an attempt to conceal its true behavior from an analyst using the debugger. For any given artifact, there can be multiple *artifact mitigation strategies* for preventing exposure of the artifact to the sample.

Although it is possible to obtain certain artifacts of stealthy malware statically (e.g., strings), malware samples may employ obfuscation or packing techniques which limit static approaches. Static analysis can also be prone to false positives and does not provide execution behavior of a sample to be leveraged in the analysis of future samples.

Each artifact mitigation strategy (i.e. specific handling of an observed artifact) comes with an associated *mitigation cost*, which captures overhead, development effort, or other economic disadvantage, and *artifact coverage*, which is the fraction of stealthy samples defeated by the strategy.

Earlier work [31], [45] used observed differences between runs in disparate environments to determine which artifacts are used by a stealthy malware sample. Historically, however, such approaches have not involved many analysis environments, instead focusing on case studies that compare runs between limited numbers of virtualization environments. Given the growing number of malware mitigation

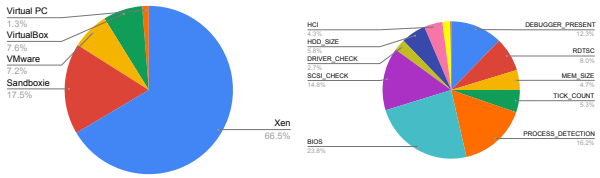

Fig. 1: VM process names (left) and artifacts detected by samples of the stealthy malware dataset used to create MIMOSA's configurations (right).

techniques, there is a need for techniques that enable fine-grained control over the artifacts exposed by the analysis environment. By precomputing a set of configurations that can be tested in parallel and reused for new malware samples, we hypothesize that MIMOSA will both increase coverage and analysis scalability of stealthy malware compared to existing automated malware analysis techniques.

Machine Learning in Malware Analysis Machine learning has recently garnered attention for automatic malware analysis, particularly for classifying malware samples [46]. Previous works used classical machine learning algorithms such as decision trees [47] and clustering [48], but recent literature has leveraged deep learning to combat the deluge of new malware variants. Specifically, prior work has seen successful performance on representations of the raw binary as byte sequences [49] and byte images [50]. For our framework, we use deep learning to automatically assign malware binaries to a feasible configuration for increased scalability (i.e., to avoid running each sample in every configuration). Past classifiers have seen success in determining whether or not a sample is malicious [48], [50], [51], but there is limited actionable information from this result with respect to individual sample behavior. Here, we use machine learning to predict which configuration is expected to execute an input sample, which provides richer information about each sample's behavior with reduced human effort when it executed on a predicted configuration. Naeem *et al.* [52] uses computer vision to classify representative images from 10,000 malware binaries to up to 9 malware families. However, these samples are not stealthy and the approach does not directly enable collection of per-sample behavioral information in an automated analysis setting.

3 PROPOSED WORKFLOW

In this section, we describe the MIMOSA framework that automatically analyzes stealthy malware efficiently. Current techniques either rely on human creativity (e.g., debugging with IDA Pro [53] or OllyDbg [54]) or heavy-weight analysis techniques that incur substantial overhead (e.g., MalT [30], Ether [14], Ghidra [55], or LO-PHI [24]). Moreover, differencing approaches, such that of as Balzarotti *et al.* [31], execute a sample in multiple instrumented environments and use the difference in runs to determine which artifact is used by the sample, potentially wasting resources.

Given (1) a list of available artifacts, (2) strategies available for mitigating them, and (3) a cost model of mitigation strategies, MIMOSA's objective is to select a small set of configurations that cover as many stealthy malware samples as possible[2]. Predicted configurations can then be deployed on a fixed number of servers for efficient analysis of stealthy malware samples.

TABLE 1: Taxonomy of 9 VM artifacts detected by stealthy malware [30] that motivated MIMOSA configurations.

Artifact Name	Artifact Description
Hardware ID	VMs with obvious strings (e.g., "VMWare Drive") or identifiers (e.g., MAC address).
Registry Key	Windows VMs have telling registry keys (e.g., dates and times of VM creation).
CPU behavior	VMs may not faithfully reproduce CPU instructions.
Resource constraint	Analysis VMs may have sparse resources (e.g., <20GB disk)
Timing	VMs may not virtualize internal timers, or may incur noticeable overhead
Debugger Presence	Tools like `gdb` that instrument samples are detectable
API Calls	API calls hooked for analysis may be detectable.
Process Names	VM monitoring processes have specific names (e.g., vmtoolsd in VMWare).
HCI	check the human interactions with system (e.g. keyboard activity)

Broadly, the configurations shown in Table 2 were constructed from in-depth analysis of the original dataset of 1535 stealthy malware samples, including execution traces and process activity. The breakdown of artifacts and VMs detected from this dataset is shown in Figure 1. Given the relatively small number of VM backends, coupled with the artifact taxonomy in Table 1, we intuit that our initial MIMOSA configurations can effectively generalize to new stealthy malware samples. This is borne out in Section 6.3.

For a new malware sample that is suspected but not known with certainty to exhibit stealthy behavior (e.g., using signature matching or other static heuristics), MIMOSA predicts the configuration best suited for the sample to execute in based on a low-cost, high-coverage subset of configurations provided by our covering algorithm. Each subset of configurations is meant to mitigate a different specific subset of artifacts. Next, depending on available resources, the sample is scheduled to run on one (or optionally more) predicted configurations. After attempting to execute the sample, we examine analysis logs to determine if the sample executed without detecting analysis. If the sample executed, we record its behavior and proceed with the next sample. Otherwise, we select the next best configuration based on the classifier prediction until the sample executes. If a sample does not execute successfully on any available config-

2. that is, the malware sample runs without detecting environment instrumentation.

uration, it must be manually analyzed to understand the aspects of stealthy behavior that are not mitigated by extant configurations. In this way, MIMOSA can continually learn and incorporate correct predictions into subsequent decisions about unseen evasive malware samples. This offers the benefit of reduced human effort required to analyze new stealthy samples while also ensuring efficient use of analysis resources. Additionally, by identifying configurations where suspected stealthy malware samples will faithfully execute, MIMOSA offers an improvement over static techniques by collecting specific behaviors during program execution.

MIMOSA's workflow is illustrated in Figure 2. In Step 1, pre-processing for a new malware corpus works by manually reverse engineering a random subset of samples to obtain artifacts and corresponding mitigations. In this case, new samples are fed to the trained classifier to predict configurations in which to execute the sample. In Step 2, we apply our covering algorithm, which takes (1) a list of artifacts, (2) corresponding costs to mitigate each artifact (Section 4.1), and (3) a set of mitigation strategies for each artifact (Section 4.2) as input. The algorithm returns a Pareto frontier of covering sets of configurations for designing and deploying different analysis environments. Each covering is represented as a vector of bits, where each element indicates whether that artifact is mitigated in the environment's configuration. The cost model can include a multitude of factors including VM run-time, memory usage, mitigation development time, etc.

Next, these coverings are realized in a malware analysis cluster by configuring specific computing resources. We use VMCloak [21] to provision VMs within our framework to communicate with the Cuckoo agent [56] during the analysis of a sample in order to collect behavioral artifacts.

Coverings generated by MIMOSA mitigate commonly-used artifacts by stealthy malware samples (e.g., using QEMU or VirtualBox as a virtualization backend, hooking certain API calls like `isDebuggerPresent`, or using specific kernel modules or drivers).

Each configuration can be deployed on one or more servers within the analysis cluster. We assume access to a suite of hypervisors and hardware resources that can be configured a priori to realize the set of mitigations specified by the covering. We implemented 13 such hypervisor and hardware configurations (Table 2), managed by our Dispatcher module to spin up and spin down analysis resources as samples are processed.

In Step 3, the trained classifier predicts the configuration(s) in which to run the sample based on the artifacts likely to be engaged by the sample. This classifier is trained from a small subset of the malware corpus that we assume has been manually analyzed to reveal defects and may either be used out of the box or after finetuning on a subset of new samples, depending on desired performance.

Each sample in turn is then executed as a process within the predicted configuration using Cuckoo Sandbox [56]. As the process runs, MIMOSA collects API traces and VM state logs through Virtual Machine Introspection (VMI). In Step 4, we analyze the logs and JSON report generated by the Cuckoo agent during sample execution to determine whether the sample executed. These heuristics include, for example, process exit codes, process names, process lifetimes, file operations, loaded libraries, API calls, and registry key modifications.

We conclude that a sample has faithfully executed (i.e., it does not detect its environment) if it contains "behavior" data in the JSON report generated by the Cuckoo analysis agent but does not trigger the heuristic detection (i.e., evasive behavior is mitigated). For example, a sample that calls `GetSystemInfo` and receives a cpu count less than 2 has detected the analysis environment according to the cpu detection heuristic. Similarly, a sample that calls any of `GetFileAttributesW`, `GetFileAttributesA`, `NtCreateFile` with a VM driver (e.g. `Vboxservice.exe`) as an argument has detected the analysis environment according to the driver detection heuristic. The detection heuristics are extensible but not perfect; MIMOSA prioritizes the performance improvement of accelerated analysis over potential false positives and negatives from heuristic detection.

As a concrete example, the sample with MD5 hash beginning with `0f3ec573` was predicted to execute on the **vbox_2** configuration (see Table 3) but in fact detects the presence of the `VBoxGuest` driver using `NtCreateFile` (indicated in the **behavior["processes"]** entry of the JSON report) and tries to read its file attributes. Here, we conclude the sample fails to execute since it identifies the presence of `VBoxGuest` according to the driver detection heuristic. This is corroborated by a signature match in the same report which indicates that this sample "Detects VirtualBox through the presence of a device".

If a sample's analysis report does not contain a "behavior" entry, then we predict the next best configuration and run the sample on it. If a sample does not execute in any available analysis environment, then it is reserved for manual analysis, and can be fed back to MIMOSA to create a new VM configuration that mitigates similar artifacts, and to finetune the classifier as needed.

Section 6 uses a well-labeled corpus of 1535 stealthy malware samples to evaluate MIMOSA's effectiveness as well as another, unlabeled corpus of 1221 BluePill stealthy samples [41] to demonstrate that MIMOSA can generalize to unseen malware samples. We next describe our approach for producing covering sets of VM configurations.

4 FINDING COVERING SETS

A key insight of our approach is that efficient analysis of malware samples must balance the number of artifacts that are mitigated and the cost of deploying multiple mitigations. Because stealthy malware uses artifacts to evade detection, we desire mitigating as many artifacts as possible to minimize the likelihood the sample detects the analysis environment. However, mitigating all artifacts simultaneously imposes unreasonable costs or is impossible entirely, so instead the goal is to find sets of *configurations*, where each configuration implements a subset of the available artifact mitigations. Then, each configuration can be run simultaneously and relatively inexpensively while collectively defeating most of the stealthy malware samples.

Given a set of artifact mitigation strategies (configurations) and a model that assigns a cost to each strategy, we describe an algorithm for efficiently selecting a set that

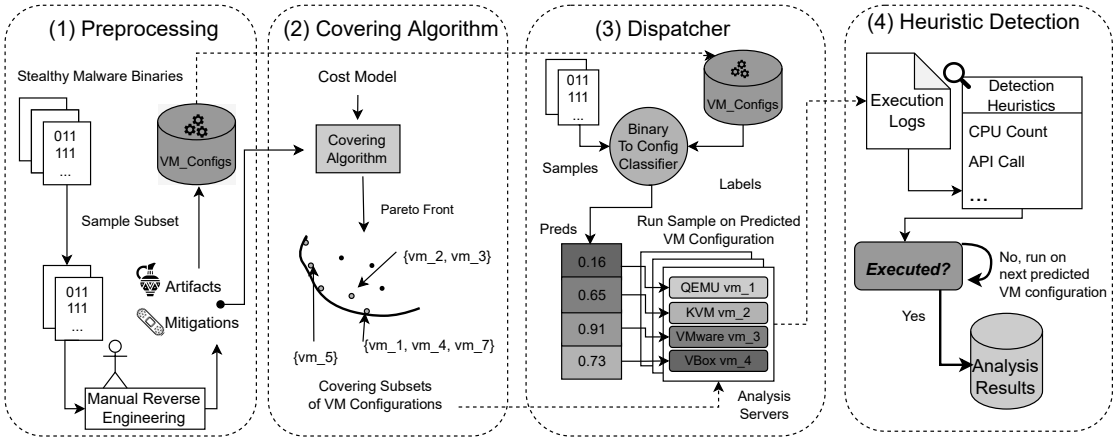

Fig. 2: Overview of the MIMOSA workflow: In step (1), we identify a set of known artifacts and mitigations based on manual reverse engineering of a random subset of new samples from a corpus and a cost model. In step (2), our covering algorithm generates a Pareto front of low cost, high coverage configuration sets to analyze stealthy malware samples. In step (3), our classifier assigns malware samples to configurations on which each sample should execute, and the corresponding VMs are dispatched to designated servers for analysis. In step (4), heuristic analysis indicates whether the malware sample detected its analysis in the VM instance or not, based on behavioral information in the analysis report.

TABLE 2: MIMOSA's configurations and the artifacts that they mitigate. Each column indicates if a specific artifact is mitigated in that configuration.

Index	Backend	Configuration	Process	Debugger	CPUID	RDTSC	CPU #	Invalid Inst.	TickCount	HCI	BIOS	File Check	HDD - SCSI	Disk size	Memory	MAC	ACPI
1	KVM	kvm_patched_1	✓	✓	–	–	✓	✓	–	✓	✓	–	–	–	–	✓	✓
2		kvm_patched_2	✓	✓	–	–	✓	✓	–	✓	✓	–	✓	–	✓	–	✓
3	VMWare	vmware_3	✓	✓	–	–	✓	✓	–	✓	✓	–	–	–	–	✓	✓
4		vmware_2	✓	✓	–	–	–	–	–	✓	–	✓	–	–	✓	–	✓
5		vmware_2_vmtools	–	✓	–	–	✓	–	–	✓	–	–	–	–	✓	–	✓
6	KVM	kvm_legacy_1	✓	✓	–	–	✓	✓	–	✓	✓	✓	–	–	✓	✓	✓
7		kvm_legacy_2	✓	✓	–	–	–	✓	–	✓	✓	✓	–	–	–	–	✓
8	Virtualbox	vbox_1_guestadditions	–	✓	–	–	–	✓	–	✓	✓	–	✓	✓	–	–	–
9		vbox_2_guestadditions	–	✓	–	–	✓	✓	–	✓	–	–	✓	✓	–	✓	✓
10		vbox_1	✓	✓	✓	–	✓	✓	–	✓	✓	✓	✓	✓	✓	–	–
11		vbox_2	✓	✓	–	–	–	✓	–	✓	–	✓	✓	✓	–	✓	–
12	QEMU	qemu_legacy_1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	–	–	✓	✓
13		qemu_legacy_2	✓	✓	✓	–	✓	✓	–	✓	✓	✓	✓	–	✓	–	✓

maximizes coverage while minimizing cost. There are three components to consider which are outlined below:

1) Any non-negative **cost model**. For example, development time plus analysis speed. In our evaluation, we consider cost as the runtime overhead of a VM configuration in terms of disk throughput, CPU time, and available memory during analysis.
2) For each type of artifact, there is a **mitigation strategy** which prevents the malware from detecting its environment using that artifact, and a cost associated with realizing the mitigation strategy.
3) The **covering algorithm** then selects from the many possible configurations to produce a set that optimizes the trade-off between cost and coverage.

4.1 Cost Model

Abstractly, we model cost as a function mapping each artifact mitigation strategy to the nonnegative real numbers, $\mathbb{R}_{\geq 0}$. Our approach operates regardless of how this cost function is defined, and may readily be modified by an analyst. If a mitigation strategy is known (e.g., from a published paper) but an implementation is not available, the analysis organization incurs a software development cost to implement it. For example, a full QEMU-emulated system may take longer to boot and execute than a Virtualbox configuration but may defeat samples that detect Virtualbox registry keys.

Given an available set of implemented mitigations, a second cost is the overhead of deploying them. There are a number of relevant metrics to consider here such as throughput and energy consumption.

Given the rate at which new stealthy samples are discovered [39], [45] and the costs associated with zero-day exploits, rapid analysis response is often paramount. Given a fixed computing budget, if one approach admits analysis after 100 time units and another approach only admits analysis after 800 time units, the former is preferred. For example, consider a scenario in which ten servers are available. One could deploy heavyweight tools (such as Ether [14], BareCloud [45], or MalT [30]) on all ten servers; this would produce a suitable analysis but is not efficient: it would take a long time for an analysis to run to completion. Alternatively, one could deploy lighter-weight systems such as LO-PHI [24] or VMI. This would be more efficient, but risks detection by samples in the input corpus, at which point the analysis fails.

4.2 Selecting Artifacts and Mitigations

First, we identify a number of potential artifacts commonly used by our corpus of stealthy malware samples (Section 6.1). We followed existing literature [30], [40] and the pafish tool [57] to group these artifacts into a taxonomy of categories. We consider 9 artifact families (Table 1) and 34 total specific artifacts, which together represent behavior indicative of stealthy malware.

For each artifact, we implemented several mitigation strategies across a number of hypervisor backends. The mitigation strategies ranged in complexity from straightforward scripting (e.g., synthetic mouse movements) to more complex patches to the hypervisor source code (e.g., to hook kernel API calls made within the guest).

As new artifacts are discovered in the future and exploited by adversaries, mitigations can be implemented and added to MIMOSA incrementally since the cost analysis and coverings construction generalize regardless of artifact behavior or exploitation, and classifier performance will ostensibly improve with additional training data.

4.3 Generating Coverings

Next, we present our algorithm for generating a set of low-cost configurations that cover as many artifacts as possible. This algorithm is used to identify appropriate configurations from a subset of previously analyzed malware. Once identified, these configurations can be reused or updated for unseen samples, such that a single configuration can cover many previously unseen samples.

Let $\mathcal{A} = \{A_1, \cdots, A_n\}$ be the set of n artifacts, $\mathcal{C} = \{C_1, \cdots, C_s\}$ be the set of s configurations, and $\mathcal{S} = \{S_1, \cdots, S_p\}$ be the set of samples observed. For each sample S_i, we associate with it a set of *engaged artifacts* $E(S_i) \subset \mathcal{A}$. For each configuration C_j, we associate it with a set of *mitigations* $M(C_j) \subset \mathcal{A}$.

Our goal is to construct a binary 2-dimensional array (called a *covering*), where each of the rows corresponds to a configuration, and each of the columns corresponds to an artifact, with the following property. For any sample S_i, there are configurations $C_{j_1}, \cdots, C_{j_\ell}$ for which $E(S_i) \subset M(C_{j_1}) \cup \cdots \cup M(C_{j_\ell})$; in other words, for any sample, there are some configurations that together fully mitigate the sample. In terms of the array itself, suppose that $E(S_i)$ involves the columns/artifacts $e_1, \cdots, e_m$. Then, the union of all rows r in these columns contains a 1 in each entry, where 1 in column e_i indicates that configuration r mitigates the artifact e_i, and 0 otherwise. If the property is not maintained, we generate an array that mitigates as many samples as possible (high coverage), while also having the cost(s) of the chosen configurations be as low as possible.

In addition, we maintain a set of *desirably high* (DH) and *desirably low* (DL) characteristics that reflect the relative importance on the cost of implementing a particular strategy (e.g., runtime overhead is DL, while coverage is DH). In general, we want to generate a set of configurations such that the covering's DH characteristics are as large as possible, and the DL characteristics are as low as possible. Next, we walk through the algorithm.

For each configuration and artifact, we mark whether or not the configuration mitigates the artifact. Each configuration has an associated cost (e.g., runtime overhead) and coverage (e.g., the fraction of well-labeled malware samples that can be exeucted within that configuration). Next, we generate all subsets of configurations; suppose these subsets are $S_1, \cdots, S_k$. We say that a subset of configurations S_i *dominates* another subset S_j if the following properties hold:

- S_i's DH characteristics are all at least those of S_j,
- S_i's DL characteristics are all at most those of S_j, and
- either (1) some DH characteristic of S_i is strictly larger than that of S_j, or (2) some DL characteristic of S_i is strictly smaller than that of S_j.

The *Pareto front* of the subsets is the collection of subsets $\mathcal{S}$ such that none of the subsets dominates any other in $\mathcal{S}$, which can be found by *non-dominated sorting* [58].

This algorithm is not efficient because it examines every subset, which takes exponential time in the number of configurations. We give an optimization that improves the running time in practice by assuming that all of the DH and DL characteristics are *monotonic*, which means that if a new configuration c is added to a set of configurations $\mathcal{S}$, then $\mathcal{S} \cup \{c\}$ cannot have larger DH characteristics nor smaller DL characteristics than those of $\mathcal{S}$. For example, adding a configuration does not decrease the total deployment time, so this is a monotonic DL characteristic. Meanwhile, coverage (defined in Section 2) is monotonically increasing because adding configurations does not decrease overall coverage. In other words, introducing a new configuration for a sample to run on will in the worst case not increase coverage. Coverage of samples cannot be reduced by adding analysis configurations to any given subset. Formally, let $\mathcal{A}_i$ be all subsets of size i. Let a be a subset in $\mathcal{A}_i$, and let c be any configuration not in a. If the coverage of $a \cup \{c\}$ is more than a, then we need to observe some subset in $\mathcal{A}_{i+1}$ (because $a \cup \{c\}$ is one such subset). However, if the coverage does not increase for any subset in $\mathcal{A}_i$ with any new configuration c, then we can terminate the algorithm because (1) the coverage does not increase, and (2) the characteristics are monotone.

An advantage of our algorithm is that it is highly likely that a configuration will cover the artifacts employed by any observed sample since the algorithm produces minimal subsets of configurations (i.e., deleting any configuration from any subset will cause coverage to decrease).

5 Machine Learning Approach

Recall that the MIMOSA approach requires predicting a configuration in which a sample is likely to execute successfully. To achieve this, we build an image classifier that takes as input a grayscale image representation of a single raw sample binary, similar to previous work on malware classification [59], [60], [61], [62], [63]. We use our labeled 1535 sample dataset to train a classifier whose output predictions represent model confidence that a given sample will execute successfully on an available configuration (i.e., multi-label classification). In practice, we expect this step to require in the worst case an initial small, random subset of samples to manually reverse engineer to produce new configurations and finetune the classifier. This is only required in the event that a new corpus of malware does not execute on the configurations proposed in this work. The remaining samples (i.e., those not manually reverse engineered) in the

corpus can then be fed to this classifier to predict the best new configuration in which to execute an unseen sample.

Model Selection The `Malware2Config` model assigns a sample binary as a grayscale image to a configuration represented by a binary array. We consider ResNet [64], ResNeXt [65], and ConvNeXt [66] architectures. ResNet and ResNeXt are established vision models for image classification that use residual connections, whereas ConvNeXt leverages self-attention to consider information from disparate locations in the input image. We use sigmoid output layers and cross-entropy loss to provide a probability for each predicted configuration. We also explore 2 output layer variants: a fully-connected dense layer that directly computes the prediction vector ("dense") vs. N branches that receive identical copies of the final hidden layer and compute class belonging with a shallow binary classifier ("branch"). These models allow us to rapidly predict the best suitable configuration in which to run a given malware sample by considering the sample as a grayscale image modulated by raw byte values.

Model Performance To maximize model performance, we leverage RayTune [67] to conduct a continuous random hyperparameter search over the following values: image size (64x64 to 512x512), epochs (10 to 100), learning rate (log uniform from 0.00005 to 0.01), validation percentage (0.01 to 0.3), output layer variant ("dense" or "branch"), and optimizer (Adam, SGD, or RMSProp). Our highest performing model uses the ResNeXt architecture that is 50 layers deep with a "dense" output layer, input image size of 64x64, batch size of 16, learning rate of 0.001, binary cross-entropy loss, and Adam optimizer. This model *was not* pre-trained on the CIFAR10 dataset and achieved an average validation hit rate of 96%.

6 Empirical Evaluation

MIMOSA adapts coverings to choose artifact mitigation strategies that enable the accurate and efficient analysis of stealthy malware that would otherwise require considerable human effort to analyze and understand. In this section, we present results from empirical evaluations of MIMOSA.

We begin by introducing an indicative use case. Consider an enterprise that desires to use a set of servers with finite capacity for automated malware classification and triage. We assume that low-latency analysis of stealthy samples is paramount: given a fixed set of computing resources, we want the analysis of a given sample to complete as quickly as possible to support subsequent human analysis, defense creation, signature generation, etc. We further assume that the input samples are stealthy, and the analysis tool must mitigate the artifacts exposed to each sample to prevent subversion. Although it might be possible to use all servers available to the enterprise to mitigate all potential artifacts, this is not an efficient use of resources and does not provide the lowest analysis latency. Instead, we apply our covering algorithm (Section 4.3) to determine which sets of artifacts are to be mitigated by each server to minimize analysis latency while also maximizing coverage. Again, we assume that a subset of the stealthy malware corpus of interest has been manually reverse engineered to create covering configurations and to train the classifier. To evaluate our approach, we consider three research questions:

RQ1 Coverage — Does MIMOSA produce artifact mitigation configurations that effectively cover stealthy malware samples?

RQ2 Generalizability — Does MIMOSA effectively predict configurations for new evasive malware samples to execute successfully on existing configurations?

RQ3 Scalability — What tradeoffs exist in scaling MIMOSA to a large evasive malware dataset given a small number of representative samples?

We first discuss the corpus of malware we used in our evaluation. Then, we discuss each research question in turn.

6.1 Malware Corpus Selection

We consider stealthy malware that targets Windows. Of the many available malware corpora, only a few focus directly on stealthy malware, in part because they are so difficult to analyze automatically.

Instead, we obtained a set of 1535 unique samples from independent security researchers, which are analyzed according to the artifacts they use. A breakdown of these is shown in Figure 1. This dataset contains malware samples that have been manually identified as stealthy and precisely curated and fully reverse engineered to obtain behavioral information from execution traces. Other work has used larger malware databases for similar experiments [68], [69], but as mentioned above these datasets are not labeled with enough specificity for our study. Additionally, we include analysis outcomes for 1221 stealthy samples from the Bluepill [41] dataset based on the MIMOSA classifier predictions to further demonstrate the generalization ability of MIMOSA's original configurations.

6.2 RQ1: Coverage — Artifact Mitigation

In this experiment, MIMOSA deploys artifact mitigation strategies to analysis servers. We define the *configuration set size* as the number of configurations combined together—this is an input parameter that represents the number of distinct configurations that the user is willing to run concurrently on a set of analysis servers.

For example, if more servers are available for analysis, a larger configuration set size can be selected. We note that the configuration set size is a distinct quantity from the number of servers available for analysis; for some corpora of malware it may be reasonable to duplicate a single high coverage configuration across multiple servers while for others a different configuration is deployed on each available server. We say that a stealthy malware sample is successfully analyzed if at least one configuration in the configuration set produced by MIMOSA mitigates all of the artifacts used by that sample.

For each configuration, we represent artifact coverage as a binary array in which each set bit implies that that particular artifact has been successfully mitigated in the environment. Table 2 gives details about each configuration.

We use VMWare, VirtualBox, KVM, and QEMU backends for virtualizing guests to complete an analysis of each sample. We use 13 different configurations across

these backends for conducting analyses. Each configuration implements a subset of mitigations against each class of artifacts. For example, the *kvm_patched_conf1* configuration contains intentionally low RAM size ($<$ 1GB), exposing the RAM detection family of artifacts, but also contains custom patches that remove all QEMU-related hardcoded strings throughout the source (KVM accelerates QEMU emulation). In contrast, the *VMWare_2* configuration employs the VMWare Tools suite for faster execution, exposing process names (i.e., of VMWare Tools). Broadly, we designed and implemented these configurations by considering the families of artifacts exposed by our dataset (Section 6.1) and the expected complexity in mitigating each artifact family across each virtualization backend.

Next, we determine whether each evasive sample eventually executes to completion within the configuration predicted by our system. To achieve this, we collect multiple API trace logs for each sample analyzed and aggregate these traces to bridge the semantic gap [70], [71]which enables reconstruction of higher abstraction API traces invoked against the guest OS.

Given each trace of each sample, we confirm detection results based on the malware's behavior within the configuration on which the sample was predicted to execute. Specifically, we follow the malware execution trace up to the point when it starts to create, manipulate, or remove a memory section, segment, or page using APIs such as `NtCreateSection`, `NtMapViewOfSection`, or `NtSetContext`. Then, we compare these analysis results against ground truth established in our corpus to ensure that the malicious process executed completely. If the analysis differs from the ground truth (e.g., if the sample detects the environment and hides its behavior), we say that configuration does *not* cover the sample. If there exists at least one configuration that *does* cover the sample, we call that sample covered. If not, we say that the sample is not covered, implying that it must be executed on the next predicted configuration or manually analyzed. We show the coverage rate of each configuration across our entire corpus of malware in Figure 3.

6.2.1 RQ1 Result Summary

Our mitigation strategies and corresponding configurations provide varying coverage levels across an indicative dataset of 1535 stealthy malware samples, allowing us to explore the trade-off space between coverage provided by analysis tools and the cost of deploying those tools or acquiring analyses. As shown in Figure 3, the configurations cover a range of samples depending on the VM backend which is related to the artifact mitigation supplied by that configuration. The configuration *qemu_legacy_1* is able to cover 1244 samples on its own, while *vmware_2_vmtools* only covers 189 samples on its own. This demonstrates the benefit of combining configurations based on collections of artifact subsets to mitigate more artifacts and cover more samples.

6.3 RQ2: Generalizability — Analyzing New Samples

In this section, we assess the generalizability of the MIMOSA pipeline by measuring whether a new set of evasive malware samples will execute successfully on predicted configurations. Specifically, we classify each of 1221 samples from the BluePill [41] dataset to 8 of the original MIMOSA configurations. Eight configurations total were selected across KVM and VirtualBox backends since these backends had the lowest cost configurations along with QEMU. To expedite the analysis, we omit QEMU since it is similar to KVM with slower emulation. To summarize, we use a single analysis server with VirtualBox and KVM backends for this experiment. Each sample was executed according to the configuration(s) predicted by the classifier and the same detection heuristics were used to determine whether the sample executed successfully. We consider samples that detect the environment with the Human Computer Interaction (HCI) heuristic to have executed successfully in that configuration since adjustments like decreasing the time interval between automated cursor movements or increasing the area covered by the cursor (i.e., making VM activity appear more human-like) can be implemented to evade HCI detection.

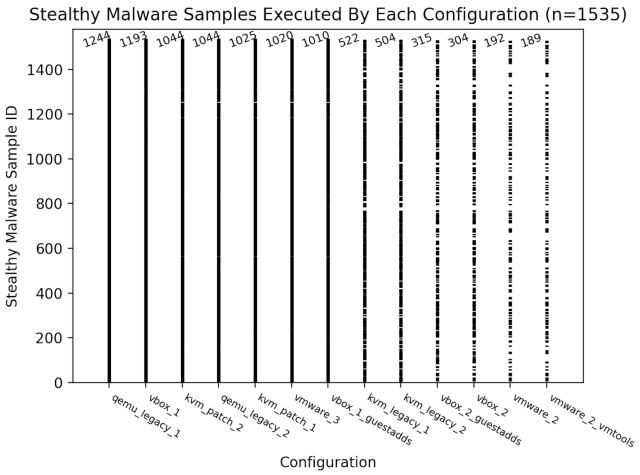

Fig. 3: Samples that executed on the original MIMOSA configurations. A gap in the bar indicates that a sample was not covered by that configuration. Further to the left are configurations with higher cost, indicating that effectively covering all samples requires multiple configurations.

The results of this study are summarized by Table 3. In total, 1207/1221 BluePill samples were successfully analyzed across the 8 available configurations using a single analysis server, leaving 14 samples to manual analysis. In other words, *nearly 99% of samples from an unseen dataset of stealthy malware samples executed successfully using a subset of the original* MIMOSA *configurations*. The remaining 14 samples contain artifacts that were not mitigated in the configurations tested, and manual reverse engineering of these samples can contribute to identifying new artifacts and additional configurations for use in subsequent automated analysis. This allows for MIMOSA to achieve monotonically increasing coverage as a greater diversity of stealthy malware is analyzed over time.

Based on our classifier predictions, 110 samples execute on the top-1 configuration predicted by the classifier, 317 on top-3, 390 on top-5, and 370 top-7, leaving only 20 samples that executed on the configuration with the lowest prediction confidence (non-cumulative).

From these results, we conclude that MIMOSA's configurations successfully generalize to new evasive malware

TABLE 3: BluePill Samples that Executed on Predicted MIMOSA Configurations (listed by decreasing cost from top to bottom, left to right)

Config	# Executed	Config	# Executed
kvm_patched_1	622	vbox_1	96
vbox_1_guestadds	179	kvm_legacy_2	21
vbox_2_guestadds	147	kvm_legacy_1	4
vbox_2	100	kvm_patched_2	38
	Total	1207/1221	

samples and can continually increase the coverage provided by the MIMOSA pipeline by using insights from manually analyzed samples to cover yet-unseen samples engaging similar artifacts.

6.4 RQ3: Scalability — Tradeoffs in Scaling MIMOSA

MIMOSA identifies Pareto-optimal coverings that provides accurate stealthy malware analyses while minimizing the resource allocation required to obtain those analyses. Although the covering set reduces the number of configurations to spin up, it does not identify the configuration in which each sample will veritably run. To prevent running all samples on every configuration in the covering set, the model predicts which configurations the samples will run on in an effort to both expedite and scale MIMOSA.

To demonstrate the model's potential effect, we simulated MIMOSA's scalability in terms of the total runtime under four scenarios: (1) **Ideal** scenario (i.e. 100% model accuracy) with weighted round-robin scheduling, (2) **Random** scheduling of samples to configurations, (3) Industry-standard king-of-the-hill (**KotH**) scheduling of all samples to the "best" configuration, and (4) **MIMOSA**'s scenario (i.e. approx. 90% model accuracy) with weighted round-robin scheduling. All scenarios were subject to load balancing after initial allocation to use all available servers and reduce the total runtime as much as possible.

For the **Ideal** and **MIMOSA** scenarios, model accuracy is the simulated success rate where a random (100 - success rate)% of samples fail to run (e.g., because the sample detects the predicted configuration during execution) and have to be rescheduled. We define *rescheduling* as subsequent iterations in which the predictions are updated to reflect prior execution failures on the previously predicted configurations so a failed sample does not rerun on the same configuration; the total runtime comprises the time to complete all rescheduling iterations.

For the random and KotH scenarios, we assume no rescheduling. One consideration is the threshold for a positive label to indicate coverage (i.e., whether the sample will execute); at a 50% threshold, our model predicts 90% of samples to run on the KotH configuration (`vbox_2`). To better reflect the coverages reported in Section 6.2.1, we threshold positive labels at a higher predictive confidence value of 70% for the random and KotH scenarios. Although this is not directly comparable to the proposed weighted round-robin procedure because random and KotH scenarios would not rely on model predictions for running samples, we use them in a best-effort attempt to make as close a comparison as possible. Figure 4 controls for the number of servers vs. runtime for each of the four scenarios.

Figure 5 shows that total runtime decreases with increased model accuracy. From these results, we observe that our proposed method approaches the ideal scenario as the number of available servers increases, suggesting that as the system is scaled up *our method holistically outperforms conventional methods in regards to runtime and coverage*. Our simulation also indicates that the industry-standard KotH method trades off coverage for runtime while our method scales with a comparable runtime while accomplishing almost perfect coverage.

7 DISCUSSION

In this section, we discuss (1) potential threats to the validity of the experimental results, (2) using MIMOSA for controlling an adaptive malware analysis system, and (3) potential future improvements that can be made to cost functions.

7.1 Threats to Validity

First, we characterized artifact families according to conceptual similarity. The artifact families ultimately inform what structure the corresponding covering takes. There is no standard method for classifying artifacts in this manner—the effectiveness or utility of MIMOSA could change depending on the specific assumptions we made about which artifacts are categorically similar.

Second, although our evaluation incorporated 1535 stealthy malware samples from the wild, we produced configurations whose costs were measured in isolation (e.g., we measured CPU utilization separately from memory utilization).

Finally, we do not evaluate the risk of adversarial perturbations to our malware image classifier [72], [73], [74]. An attacker could undetectably modify pixel values of a malware image such that our classifier mispredicts a configuration for a perturbed image sample. In the worst case, the sample will run unnecessarily on multiple configurations in the case of repeated mispredictions. However, byte-level perturbations made to the binary before it is processed into an image would likely alter the original sample and could lead to unintended program behavior.

7.2 Remarks on Adaptability

MIMOSA takes as input a set of modeled mitigation strategies and associated costs, and it produces as output a coverage-optimal, low-heuristic-cost array of strategies. This approach can be extended to adapt over time to changes in the distribution of stealthy malware. For example, if new artifacts are discovered or if the costs associated with mitigating each one changes with technology, our overall approach and algorithms will still be applicable as a tool for finding cost-optimal analysis configurations. Here, the worst case requires manual analysis effort on a small number of samples along with finetuning the classifier to assign any newly constructed configurations to samples that were not covered by previous configurations.

As a specific example, recent work leveraged "wear and tear" of virtual machine environments [75]. In essence,

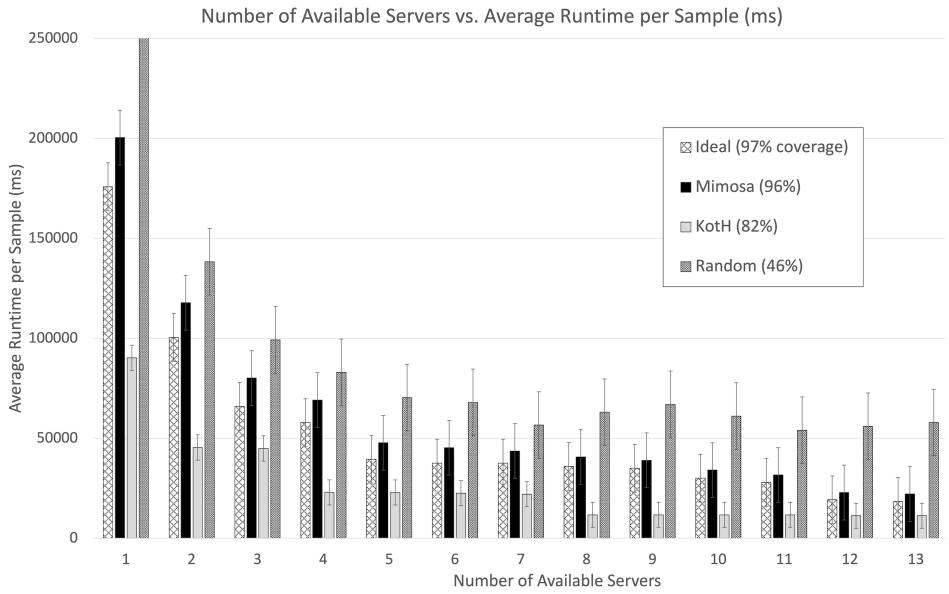

Fig. 4: Number of servers vs. total runtime for Ideal, Random, KotH, and MIMOSA scenarios. Ideal refers to weighted round-robin with a perfect model such that all samples run on the first predicted configuration. Random refers to random assignment of samples to configurations with no consideration to the model's predictions. KotH refers to the industry standard of assigning all samples to the single best configuration. Our proposed method approaches the ideal runtime with our observed accuracy and with more servers available while achieving relatively high coverage. Although the KotH configuration is low cost and is expected to run faster, it does so at the cost of coverage.

malware samples can look for evidence that an environment is "aged." An analyst that spins up a vanilla VM image may fall victim to a sample that detects if the environment is pristine and newly-created. That is, the perceived "age" of the virtualized environment is the artifact.

We do not include such artifacts in our prototype coverage calculation because our dataset did not contain samples that exploited wear and tear artifacts; however, they can be readily incorporated by implementing a corresponding mitigation.

7.3 Remarks on Cost Functions

MIMOSA currently considers optimizing for cost, which can be captured in several ways: CPU utilization, memory utilization, and runtime overhead with respect to latency.

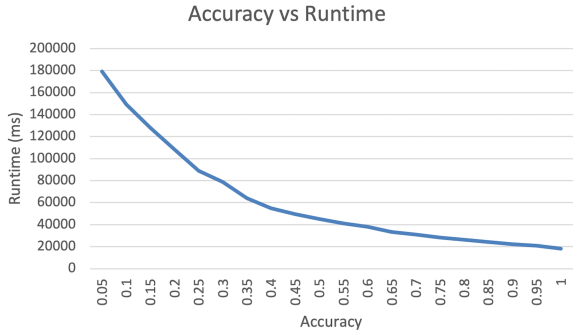

Fig. 5: Accuracy (%) vs. Total Runtime (ms) with rescheduling of MIMOSA's scenario. MIMOSA becomes more efficient as the model is more likely to accurately predict the best configuration in which to schedule each sample.

However, these one-dimensional approaches may admit coverings that are difficult to interpret. For example, in a cluster of 10 servers, assigning nine servers to do no mitigation (minimal cost) and one server to run bare metal (maximal coverage) is a well-formed solution.

We also discussed a second parameter that captures *benefit*: coverage of stealthy malware samples is important for acquiring faithful, interpretable analyses. For example, if we know a mitigation strategy will cover 90% of stealthy malware, we may be willing to pay a higher cost to use that strategy because of its overall coverage. On the other hand, a strategy that only covers 2% of stealthy malware in the wild may be disregarded.

8 RELATED WORK

Various projects have focused on detecting and evading analysis systems in both x86 executables [29], [76], [77], [78] and mobile devices (e.g., Android [79]). In this section, we discuss this work in four categories: (1) stealthy malware detection using behavioral analysis, (2) malware analysis using virtual machine infrastructure, (3) malware analysis using bare-metal machines, and (4) machine learning in malware analysis.

Stealthy Malware Detection Current stealthy malware analysis techniques generally rely either on human creativity (e.g., debugging with IDA Pro [53] or OllyDbg [54]) or heavy-weight analysis tools that incur significant overhead (e.g., MalT [30] or Ether [14]). Moreover, differing approaches, such that of as Balzarotti *et al.* [31], work by executing a sample in multiple instrumented environments and use the difference in runs to determine which artifact is used by the sample, potentially wasting resources.

Lindorfer *et al.* [80] later employed a similar technique, but used various malware sandboxes and scored their evasive behaviors. HASTEN [81] specifically focuses on stalling malware, which is a particularly difficult evasion technique to analyze because the malware appears benign for an extended period of time. TriggerScope [82] similarly examines Android programs which mask their malicious behavior until a certain *trigger* is observed. Our technique leverages a combination of multiple environments that separately mitigate different artifact families, instead providing environments in which samples are more likely to execute.

Our approach is conceptually related to SLIME [83], an automated tool for disarming anti-sandboxing techniques employed by stealthy malware. SLIME runs a sample many times, each time configuring the environment to explicitly expose certain artifacts to the sample. In contrast, our approach seeks to minimize the total cost of analysis for each sample under test. In addition, we introduce a novel application of coverings [38] that helps identify cost-optimal covering sets for stealthy malware analysis system.

Maffia *et al.* [84] provides a comprehensive analysis of over 180K Windows malware samples to understand the evolution of evasive malware behavior over the last several years. Findings indicate a roughly 10% increase in the collected samples exhibiting evasive behavior from 2016 to 2020. There is also an increased focus on Anti-Debugging tricks vs. Anti-VM checking as seen in 80% vs. 20% of recent malware samples, respectively.

Virtual Machine Analysis Ether [14] is a malware analysis framework based on hardware virtualization extensions (e.g., Intel VT). It runs outside of guest operating systems, in the hypervisor, by relying on underlying hardware features. BitBlaze [85] and Anubis [86] are QEMU-based malware analysis systems that focus on understanding malware behavior, instead of achieving better transparency. V2E [87] combines both hardware virtualization and software emulation. HyperDbg [15] uses the hardware virtualization that allows the late launching of VMX modes to install a virtual machine monitor, and run the analysis code in the VMX root mode. SPIDER [88] uses Extended Page Tables to implement invisible breakpoints and hardware virtualization to hide its side-effects. DRAKVUF [89] is another VMI-based system for both user and kernel-level analysis.

We note that recent work has investigated changes to the sandboxing environment to give it the appearance of age or use [75]. For example, a dearth of Documents, Downloads, event logs, or installed software could be a hint that the sample is not executing in a real, vulnerable environment. Although our current prototype does not address samples exhibiting such "age" checks, as discussed above, we could readily incorporate it. As with other new or yet-undiscovered artifacts, our overall framework would not change. One would simply implement a configurable mitigation against that new artifact and include it as a strategy used by our coverings algorithm.

Bare-metal Analysis BareBox [90] is a malware analysis framework based on a bare-metal machine without any virtualization or emulation techniques, which is used for analyzing user mode malware. Follow up work, BareCloud [45], uses mostly un-instrumented bare-metal machines, and is capable of analyzing stealthy malware by detecting file system changes. Willems *et al.* [91] propose a method for using branch tracing, implemented on a physical CPU, to analyze stealthy malware. LO-PHI [92] is a system capable of both live memory and disk introspection on bare-metal machines, which can be used for analyzing stealthy malware. MalT [30] uses System Management Mode to instrument a bare-metal system at the instruction level, exposing very few artifacts to the system. While LO-PHI and MalT both have high deployment overheads, they also expose very few artifacts to samples under test; thus, either could conceptually serve as our highest coverage (and highest cost) configuration.

Machine Learning in Malware Analysis Machine learning models are used most commonly in malware analysis to detect whether a sample is malicious given information such as a description of behavior, raw binaries, or API calls [51], [93]. For example, some methods [94], [95], [96] rely on sequential information such as process behavior, API calls, and Indicators of Compromise (IoCs) represented as N-grams, for malware detection. Sayadi *et al.* [97] propose a timeseries-based machine learning approach for detecting embedded malware and a fully convolutional network to classify potentially contaminated intervals in the time series data. Other works [98], [99] propose reinforcement learning for attacking static anti-malware engines with the insight that an agent perturbates the portable executable (PE) file in a way that preserves functionality and helps identify operations that lead to successful evasion.

In comparison to these approaches, we leverage computer vision for multi-label malware classification [59], [60], [63] to demonstrate the scalability of MIMOSA by taking as input the raw binary of an evasive malware sample and assigning it to one or more configurations under which it is expected to execute. This approach offers the potential key benefit that the evasive behavior of a sample need not be known a priori. As additional evasive malware samples are obtained, there is not a need to expend resources studying every one of these samples if a well-trained classifier can be obtained from a relatively small set of samples. In this case, human effort is saved since only a fraction of samples need to be manually analyzed to achieve acceptable classifier performance. In the best case, an entire new corpus of malware can be analyzed automatically with existing VM configurations. Otherwise, manually reversed samples can inform additional configurations and increase sample coverage over time.

9 CONCLUSION

Stealthy and obfuscated malware is expanding rapidly. As the security arms race continues, malware authors use increasingly sophisticated techniques to subvert analysis. The large volume of new malware released every year increases the urgency of robust automated analysis pipelines to identify and understand stealthy malware samples.

In this paper, we introduced coverings, a novel way of representing the problem of analyzing stealthy malware efficiently, and a prototype implementation called MIMOSA. We studied a broad set of artifacts exposed by analysis environments and the mitigation strategies required to prevent malware samples from using those artifacts to subvert

TABLE 4: Examples of 2 artifacts families mitigated in MIMOSA configurations

Artifact Family	Mitigation Examples
VM-specific Registry Keys	Hook RegOpenKeyEx API Hook RegQueryValueEx API Remove offending keys from guest (e.g., `HARDWARE\ACPI\DSDT\VBOX__`)
Internal Timing	Hook instructions that read MSRs Hook GetLastInputInfo API Hook GetTickCount API

detection. We modeled the mitigations using a partially ordered structure according to the number of artifacts mitigated and the cost associated with deploying that strategy. We developed 34 such mitigation strategies across 4 VM backends. We presented an algorithm that finds the lowest-cost selection of mitigation strategies to implement while guaranteeing high coverage of these artifacts. Finally, we empirically evaluated MIMOSA using 1535 labeled stealthy malware samples from the wild and analyze an additional 1207/1221 BluePill samples [41] using configurations derived from the original 1535 sample corpus to demonstrate generalization capabilities. We found that MIMOSA can find mitigation strategies that reduce the overhead and memory utilization associated with mitigating all artifacts considered, and can also cover new evasive malware samples using original configurations. Finally, we determine that MIMOSA scales automated malware analysis, optimizing for both runtime and coverage compared to existing approaches which relinquish increased coverage for decreased runtime.

ACKNOWLEDGMENTS

The views expressed in this article are those of the author(s) and do not reflect the official policy or position of the Department of the Army, Department of Defense, or the U.S. Government. The authors acknowledge the partial support of NSF (CCF CCF2211750, CICI 2115075), DARPA (FA8750-19C-0003, N6600120C4020), AFRL (FA8750-19-1-0501), and the Santa Fe Institute.

REFERENCES

[1] D. Emm, "It threat evolution q1 2022," https://securelist.com/it-threat-evolution-q1-2022/106513/.
[2] McAfee, "McAfee Labs Threat Report: June 2021," https://www.mcafee.com/enterprise/en-us/assets/reports/rp-threats-jun-2021.pdf.
[3] Trellix Threat Labs, "The Threat Report: Summer 2022," https://www.trellix.com/en-us/threat-center/threat-reports/jul-2022.html.
[4] IBM Security, "X-force threat intelligence index 2022," https://www.ibm.com/downloads/cas/ADLMYLAZ, 2022.
[5] Malwarebytes, "2022 threat review," https://www.malwarebytes.com/resources/malwarebytes-threat-review-2022/mwb_threatreview_2022_ss_v1.pdf.
[6] Sonicwall, "Sonicwall cyber threat report 2020," https://www.sonicwall.com/resources/2020-cyber-threat-report-pdf/.
[7] L. Zelster, "Mastering 4 stages of malware analysis," https://zeltser.com/mastering-4-stages-of-malware-analysis/, February 2015.
[8] D. Farmer and W. Venema, *Forensic Discover*. Addison-Wesley, 2005.
[9] D. Distler, *Malware Analysis: An Introduction*. SANS Institute, December 2007, available via https://www.sans.org/reading-room/whitepapers/malicious/malware-analysis-introduction-2103.
[10] P. L. Wedum, *Malware Analysis: A Systematic Approach*. Norwegian University of Science and Technology, 2008, master's Thesis: Available via https://brage.bibsys.no/xmlui//bitstream/handle/11250/261770/-1/347719_FULLTEXT01.pdf.
[11] VMWare, Inc., "Vmware server," http://www.vmware.com/products/server, 2008.
[12] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, "Xen and the art of virtualization," in *In Proceedings of the ACM Symposium on Operating Systems Principles*, 2003.
[13] Oracle, "VirtualBox," http://www.virtualbox.com, 2007.
[14] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*, 2008.
[15] A. Fattori, R. Paleari, L. Martignoni, and M. Monga, "Dynamic and Transparent Analysis of Commodity Production Systems," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*, 2010.
[16] Z. Deng, X. Zhang, and D. Xu, "Spider: Stealthy binary program instrumentation and debugging via hardware virtualization," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC'13)*, 2013.
[17] M. Auty, A. Case, M. Cohen, B. Dolan-Gavitt, M. H. Ligh, J. Levy, and A. Walters. Volatility framework - volatile memory extraction utility framework. [Online]. Available: http://www.volatilityfoundation.org/
[18] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 128–138.
[19] N. L. Petroni, J. Aaron, W. Timothy, F. William, and A. Arbaugh, "Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory," *Digital Investigation*, vol. 3, 2006.
[20] C. Guarnieri, A. Tanasi, J. Bremer, and M. Schloesser, "The cuckoo sandbox," 2012.
[21] T. Sick, "Vmcloak," https://github.com/hatching/vmcloak.
[22] S. Stefnisson, "Evasive malware now a commodity," https://www.securityweek.com/evasive-malware-now-commodity, 2018.
[23] X. Chen, J. Andersen, Z. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks (DSN '08)*, 2008.
[24] C. Spensky, H. Hu, and K. Leach, "LO-PHI: Low observable physical host instrumentation," in *Networks and Distributed Systems Security Symposium 2016 (NDSS 2016)*, San Diego, CA, February 2016, acceptance rate: 15.8%.
[25] R. Branco, G. Barbosa, and P. Neto, "Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies," in *Black Hat*, 2012.
[26] N. Falliere, "Windows anti-debug reference," http://www.symantec.com/connect/articles/windows-anti-debug-reference, 2010.
[27] D. Quist and V. Val Smith, "Detecting the Presence of Virtual Machines Using the Local Data Table," http://www.offensivecomputing.net/.
[28] E. Bachaalany, "Detect if your program is running inside a Virtual Machine," http://www.codeproject.com/Articles/9823/Detect-if-your-program-is-running-inside-a-Virtual.
[29] T. Raffetseder, C. Kruegel, and E. Kirda, "Detecting system emulators," in *Information Security*. Springer Berlin Heidelberg, 2007.
[30] F. Zhang, K. Leach, H. Wang, A. Stavrou, and K. Sun, "Using Hardware Features to Increase Debugging Transparency," in *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.
[31] D. Balzarotti, M. Cova, C. Karlberger, and G. Vigna, "Efficient detection of split personalities in malware." in *Networks and Distributed Systems Security Symposium*, 2010.
[32] Y. Oyama, "Trends of anti-analysis operations of malwares observed in api call logs," *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 1, pp. 69–85, 2018.
[33] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, "Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps." in *NDSS*, 2017.

[34] Y. Hong, Y. Hu, C.-M. Lai, S. Felix Wu, I. Neamtiu, P. McDaniel, P. Yu, H. Cam, and G.-J. Ahn, "Defining and detecting environment discrimination in android apps," in *Security and Privacy in Communication Networks*, X. Lin, A. Ghorbani, K. Ren, S. Zhu, and A. Zhang, Eds. Cham: Springer International Publishing, 2018, pp. 510–529.

[35] Q. Li, Y. Zhang, L. Su, Y. Wu, X. Ma, and Z. Yang, "An improved method to unveil malware's hidden behavior," in *Information Security and Cryptology*, X. Chen, D. Lin, and M. Yung, Eds. Cham: Springer International Publishing, 2018, pp. 362–382.

[36] R. Tanabe, W. Ueno, K. Ishii, K. Yoshioka, T. Matsumoto, T. Kasama, D. Inoue, and C. Rossow, "Evasive malware via identifier implanting," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Giuffrida, S. Bardin, and G. Blanc, Eds. Cham: Springer International Publishing, 2018, pp. 162–184.

[37] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis, "Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 1009–1024.

[38] S. Xu, H. kou Miao, and H. Gao, "Test suite reduction using weighted set covering techniques," *2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pp. 307–312, 2012.

[39] R. Tanabe, W. Ueno, K. Ishii, K. Yoshioka, T. Matsumoto, T. Kasama, D. Inoue, and C. Rossow, "Evasive malware via identifier implanting," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2018, pp. 162–184.

[40] A. Bulazel and B. Yener, "A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web," in *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*. ACM, 2017, p. 2.

[41] J. Rutkowska, "Blue Pill," http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html, 2006.

[42] "UPX: The Ultimate Packer for eXecutables," https://upx.github.io, retrieved November 2016.

[43] "ASPack," http://www.aspack.com, retrieved November 2016.

[44] Reversing Labs, "RLPack," https://reversinglabs.com, retrieved November 2016.

[45] D. Kirat, G. Vigna, and C. Kruegel, "Barecloud: Bare-metal analysis-based evasive malware detection." in *USENIX Security Symposium*, 2014, pp. 287–301.

[46] M. R. Smith, N. T. Johnson, J. B. Ingram, A. J. Carbajal, R. Ramyaa, E. Domschot, C. C. Lamb, S. J. Verzi, and W. P. Kegelmeyer, "Mind the gap: On bridging the semantic gap between machine learning and information security," 2020. [Online]. Available: https://arxiv.org/abs/2005.01800

[47] N. Milosevic, A. Dehghantanha, and K.-K. R. Choo, "Machine learning aided android malware classification," *Computers & Electrical Engineering*, vol. 61, pp. 266–274, 2017.

[48] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *Journal in computer virology*, vol. 7, no. 4, pp. 233–245, 2011.

[49] B. Athiwaratkun and J. W. Stokes, "Malware classification with lstm and gru language models and a character-level cnn," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017, pp. 2482–2486.

[50] L. Liu, B.-s. Wang, B. Yu, and Q.-x. Zhong, "Automatic malware classification and new malware detection using machine learning," *Frontiers of Information Technology & Electronic Engineering*, vol. 18, no. 9, pp. 1336–1347, 2017.

[51] I. Incer, M. Theodorides, S. Afroz, and D. Wagner, "Adversarially robust malware detection using monotonic classification," 03 2018, pp. 54–63.

[52] M. Naeem, R. Amin, S. Alshamrani, and A. Alshehri, "Digital forensics for malware classification: An approach for binary code to pixel vector transition," *Computational Intelligence and Neuroscience*, vol. 2022, pp. 1–12, 04 2022.

[53] IDA Pro, www.hex-rays.com/products/ida/.

[54] O. Yuschuk, "OllyDbg," www.ollydbg.de.

[55] T. N. S. Agency, "Ghidra: A software reverse engineering (sre) suite," https://ghidra-sre.org/.

[56] C. Guarnieri, "Cuckoo sandbox," https://cuckoosandbox.org/.

[57] A. Ortega, "Paranoid fish." [Online]. Available: http://github.com/a0rtega/pafish

[58] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[59] A. Bensaoud, N. Abudawaood, and J. Kalita, "Classifying malware images with convolutional neural network models," 2020. [Online]. Available: https://arxiv.org/abs/2010.16108

[60] A. Migdady, L. Smadi, and Q. Yaseen, "A cnn and image-based approach for malware analysis," in *2022 International Conference on Emerging Trends in Computing and Engineering Applications (ETCEA)*, 2022, pp. 1–6.

[61] L. Chen, "Deep transfer learning for static malware classification," 2018.

[62] N. Bhodia, P. Prajapati, F. D. Troia, and M. Stamp, "Transfer learning for image-based malware classification," 2019.

[63] M. Kalash, M. Rochan, N. Mohammed, N. D. B. Bruce, Y. Wang, and F. Iqbal, "Malware classification with deep convolutional neural networks," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2018, pp. 1–5.

[64] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[65] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," 2016. [Online]. Available: https://arxiv.org/abs/1611.05431

[66] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A convnet for the 2020s," 2022. [Online]. Available: https://arxiv.org/abs/2201.03545

[67] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, "Tune: A research platform for distributed model selection and training," *arXiv preprint arXiv:1807.05118*, 2018.

[68] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering." in *NDSS*, vol. 9. Citeseer, 2009, pp. 8–11.

[69] B. Cheng, J. Ming, J. Fu, G. Peng, T. Chen, X. Zhang, and J.-Y. Marion, "Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 395–411. [Online]. Available: https://doi.org/10.1145/3243734.3243771

[70] A. Saberi, Y. Fu, and Z. Lin, "Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[71] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion, "Sok: Introspections on trust and the semantic gap," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 605–620.

[72] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," 2015.

[73] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial perturbations against deep neural networks for malware classification," 2016.

[74] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," 2015.

[75] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis, "Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 1009–1024.

[76] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE, 2008, pp. 177–186.

[77] J. Rutkowska, "Red Pill," http://www.ouah.org/Red_Pill.html.

[78] D. Quist, V. Smith, and O. Computing, "Detecting the presence of virtual machines using the local data table," *Offensive Computing*, 2006.

[79] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu, "Morpheus: Automatically generating heuristics to detect android emulators," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. New York, NY, USA: ACM, 2014, pp. 216–225. [Online]. Available: http://doi.acm.org/10.1145/2664243.2664250

[80] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, "Detecting environment-sensitive malware," in *Recent Advances in Intrusion*

Detection, R. Sommer, D. Balzarotti, and G. Maier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 338–357.

[81] C. Kolbitsch, E. Kirda, and C. Kruegel, "The power of procrastination: detection and mitigation of execution-stalling malicious code," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 285–296.

[82] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "TriggerScope: Towards Detecting Logic Bombs in Android Apps," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, May 2016.

[83] Y. Chubachi and K. Aiko, "Slime: Automated anti-sandboxing disarmament system," https://www.blackhat.com/docs/asia-15/materials/asia-15-Chubachi-Slime-Automated-Anti-Sandboxing-Disarmament-pdf, 2015.

[84] L. Maffia, D. Nisi, P. Kotzias, G. Lagorio, S. Aonzo, and D. Balzarotti, "Longitudinal study of the prevalence of malware evasive techniques," *CoRR*, vol. abs/2112.11289, 2021. [Online]. Available: https://arxiv.org/abs/2112.11289

[85] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08)*, 2008.

[86] Anubis, "Analyzing unknown binaries," http://anubis.iseclab.org.

[87] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin, "V2E: Combining hardware virtualization and software emulation for transparent and extensible malware analysis," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'12)*, 2012. [Online]. Available: http://doi.acm.org/10.1145/2151024.2151053

[88] Z. Deng, X. Zhang, and D. Xu, "Spider: stealthy binary program instrumentation and debugging via hardware virtualization," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 289–298.

[89] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system," in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014, pp. 386–395.

[90] D. Kirat, G. Vigna, and C. Kruegel, "BareBox: Efficient malware analysis on bare-metal," in *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC'11)*, 2011.

[91] C. Willems, R. Hund, A. Fobian, D. Felsch, T. Holz, and A. Vasudevan, "Down to the bare metal: Using processor features for binary analysis," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 189–198.

[92] C. Spensky, H. Hu, and K. Leach., "LO-PHI: Low Observable Physical Host Instrumentation," in *Proceedings of 2016 Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[93] D. Gibert, C. Mateu, and J. Planes, "The rise of machine learning for detection and classification of malware: Research developments, trends and challenges," *Journal of Network and Computer Applications*, vol. 153, p. 102526, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804519303868

[94] S. Lu, Q. Li, and X. Zhu, "Stealthy malware detection based on deep neural network," *Journal of Physics: Conference Series*, vol. 1437, no. 1, p. 012123, jan 2020. [Online]. Available: https://doi.org/10.1088/1742-6596/1437/1/012123

[95] F. O. Catak, A. Yazi, and O. Elezaj, "Deep learning based sequential model for malware analysis using windows exe api calls," *PeerJ Computer Science*, vol. 6, 07 2020.

[96] M. Ali, S. Shiaeles, G. Bendiab, and B. Ghita, "Malgra: Machine learning and n-gram malware feature extraction and detection system," *Electronics*, vol. 9, no. 11, 2020. [Online]. Available: https://www.mdpi.com/2079-9292/9/11/1777

[97] H. Sayadi, Y. Gao, H. Mohammadi Makrani, T. Mohsenin, A. Sasan, S. Rafatirad, J. Lin, and H. Homayoun, "Stealthminer: Specialized time series machine learning for run-time stealthy malware detection based on microarchitectural features," in *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, ser. GLSVLSI '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 175–180. [Online]. Available: https://doi.org/10.1145/3386263.3407585

[98] H. S. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth, "Learning to evade static pe machine learning malware models via reinforcement learning," 2018. [Online]. Available: https://arxiv.org/abs/1801.08917

[99] T. Quertier, B. Marais, S. Morucci, and B. Fournel, "Merlin – malware evasion with reinforcement learning," 2022. [Online]. Available: https://arxiv.org/abs/2203.12980

Michael Sandborn received a B.S. in computer science and applied mathematics at Vanderbilt University in 2020 and is currently a computer science PhD student at Vanderbilt University. His research interests include cyber-physical systems, computer security, and anti-counterfeiting techniques.

Zach Stoebner received a B.S. in computer science and neuroscience in May 2021 and is pursuing an M.S. in computer science, both at Vanderbilt University. His research interests include deep learning, sensing & imaging, and computer vision.

Westley Weimer is a professor of Computer Science and Engineering at the University of Michigan. His main research interests include static, dynamic, and medical imaging-based techniques to improve program quality, fix defects, and understand how humans engineer software. He received a BA degree in computer science and mathematics from Cornell and MS and PhD degrees from Berkeley.

Stephanie Forrest is a professor of Computer Science at Arizona State University, where she directs the Biodesign Center for Biocomputation,Security and Society. Her interdisciplinary research focuses on the intersection of biology and computation, including cybersecurity, software engineering, and biological modeling.

Ryan Dougherty is an assistant professor in the Department of Electrical Engineering and Computer Science at West Point. He earned his B.S. and Ph.D. from Arizona State University in 2019, and was a Visiting Assistant Professor at Colgate University before joining West Point. His academic interests include software engineering, theory of computation, and combinatorial design theory.

Jules White is an associate professor of Computer Science in the Department of Computer Science at Vanderbilt University as well as the associate dean for Strategic Learning Programs. His research focuses on securing, optimizing, and leveraging data from mobile cyber-physical systems.

Kevin Leach is an assistant professor of computer sciecne at Vanderbilt University. He earned the PhD from the University of Virginia in 2016. His research combines the areas of systems security and software engineering — he has developed techniques for transparent system introspection, kernel hotpatching, and stealthy malware analysis.