Cubismo: Decloaking Server-side Malware via Cubist Program Analysis

Abbas Naderi-Afooshteh, Yonghwi Kwon, Anh Nguyen-Tuong, Mandana Bagheri-Marzijarani, and Jack W. Davidson
Department of Computer Science, University of Virginia
{abiusx,yongkwon,nguyen,mb3wz,jwd}@virginia.edu

ABSTRACT

Malware written in dynamic languages such as PHP routinely employ anti-analysis techniques such as obfuscation schemes and evasive tricks to avoid detection. On top of that, attackers use automated malware creation tools to create numerous variants with little to no manual effort.

This paper presents a system called Cubismo to solve this pressing problem. It processes potentially malicious files and decloaks their obfuscations, exposing the hidden malicious code into multiple files. The resulting files can be scanned by existing malware detection tools, leading to a much higher chance of detection. Cubismo achieves improved detection by exploring all executable statements of a suspect program counterfactually to see through complicated polymorphism, metamorphism and, obfuscation techniques and expose any malware.

Our evaluation on a real-world data set collected from a commercial web hosting company shows that Cubismo is highly effective in dissecting sophisticated metamorphic malware with multiple layers of obfuscation. In particular, it enables VirusTotal to detect 53 out of 56 zero-day malware samples in the wild, which were previously undetectable.

CCS CONCEPTS

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

KEYWORDS

PHP, Security, Malware, Obfuscation, Evasion, Counterfactual Execution

ACM Reference Format:
Abbas Naderi-Afooshteh, Yonghwi Kwon, Anh Nguyen-Tuong, Mandana Bagheri-Marzijarani, and Jack W. Davidson. 2019. Cubismo: Decloaking Server-side Malware via Cubist Program Analysis. In *2019 Annual Computer Security Applications Conference (ACSAC '19),December 9–13, 2019, San Juan, PR, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3359789.3359821

1 INTRODUCTION

Web-based malware, particularly server-side malware, is one of the most prevalent security threats nowadays. 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% of those contained PHP-based, hidden backdoors [52]. Incapsula discovered that out of 500 infected websites detected on their network, the majority of them contained PHP malware [27]. Verizon's 2017 Data Breach reported that a sizable number of web server compromises are a means to an end, allowing attackers to set up for other targets [26].

This prevalence is in part because server-side malware is typically equipped with advanced anti-analysis and anti-debugging techniques such as obfuscation and metamorphism. These techniques are implemented using dynamic language features such as dynamic code generation (e.g., `eval`), creating several challenging analysis problems [25, 31].

Detecting server-side malware is a hard problem. While there are a handful of malware detection tools, most of them are signature-based tools that are ineffective in detecting polymorphic and metamorphic malware. Specifically, the signatures are extracted from known malware samples and the given target malware by aggregating particular patterns (e.g., byte patterns) and keywords found in the samples. The detectors then compare the signatures to identify malware.

To avoid detection, server-side malware uses various obfuscation techniques, such as polymorphism and dynamic code generation, to produce malware that has a different signature from the original code. Moreover, attackers leverage various tools that generate malware variants with little to no effort, also making the signature-based approaches less effective [40, 49, 55].

From our experience in analyzing various real-world malware samples, we have observed that many malware are equipped with multiple obfuscation layers (e.g., constructing calls to `eval()` dynamically) and various evasive techniques (e.g., requiring a particular input to trigger the malware). Obfuscation techniques enable malware to thwart analysis and detection by static analysis tools, while evasive techniques hide malicious behaviors behind complicated logic to prevent dynamic analysis.

Moreover, we have observed that many malware samples found in the wild turn out to be variants of the same family, but with different obfuscation techniques applied. Interestingly, while existing malware detectors are not able to detect the obfuscated malware variants, they do recognize deobfuscated malware samples, indicating that obfuscation techniques are effective at thwarting detection tools in practice.

ACSAC '19, December 9–13, 2019, San Juan, PR, USA

ACM ISBN 978-1-4503-7628-0/19/12…$15.00
https://doi.org/10.1145/3359789.3359821

As a result, we hypothesize that revealing malicious code hidden behind anti-analysis techniques such as obfuscation and metamorphism is a key challenge in detection of server-side web malware. Unfortunately, dynamic analysis or static analysis alone is not able to handle this challenge. Specifically, dynamic analysis techniques can handle obfuscations but not evasive techniques. On the other hand, while static analysis can handle evasive tricks, it has difficulty handling obfuscation.

We propose a fully automated system, Cubismo, that more effectively uncloaks sophisticated server-side malware by neutralizing anti-analysis tricks such as obfuscation and metamorphism. Specifically, we aim to resolve parts of malware that hinder analysis and detection. In particular, we identify blocks of code that are hiding their original intention (i.e., via obfuscation). We then resolve (i.e., counter) evasion tricks to expose hidden malicious logic (e.g., often by executing the code). The exposed malicious code is then used to create new malware files by replacing the decoding and dynamic execution sequence with the deobfuscated code.

Given a program, Cubismo[1] analyzes the program to break it into small pieces, reveals the real intentions of these pieces, then reassembles the revealed intentions into the original program. The reassembled program essentially depicts the target program from multiple perspectives to present diverse aspects of the target. We call this analysis *Cubist Program Analysis* as inspired by cubist art which aims to present a subject from a multitude of viewpoints to show the piece in a greater context.

Our extensive evaluation results show that Cubismo is highly effective in revealing malicious code hidden behind sophisticated obfuscation and evasive techniques, boosting the effectiveness of existing malware detectors to find 53 new zero-day malware samples. The major contributions of this research are as follows:

- A fully automated method for decloaking obfuscated code and exposing evasive malicious behavior, called Cubist Program Analysis (CPA).
- Development of a prototype system called Cubismo that employs CPA and can be integrated into existing malware detectors, resulting in more accurate malware detection.
- An evaluation of Cubismo using a large corpus of real-world website deployments that shows Cubismo is effective in enabling VirusTotal to detect 53 out of 56 zero-day malware samples found in the dataset, with no false positives.

2 BACKGROUND

In this section, we provide background information on server-side malware and state-of-the-art malware detection tools, and we explain the challenges in detecting server-side malware.

2.1 Server-side Malware

Server-side malware aims to infect a server system (e.g., a web server). PHP malware is the most prevalent form of malware that targets web servers [26]. A distinctive characteristic of PHP malware is that it requires intervention, either via a victim user browsing the infected website to trigger execution or via an attacker manually triggering the malware. PHP malware often checks a handful of conditions (e.g., checking a particular input is provided by an attacker) to decide whether to expose malicious behavior/logic. Moreover, PHP malware actively leverages PHP's dynamic language features (e.g., `eval()`) to make analysis difficult. From our study of real-world malware (§6), we find that the majority of existing malware detectors is unable to detect obfuscated malware, even if its original malware itself is already known to be malicious.

2.2 Challenges

In this section, we enumerate three major challenges in detecting web server-side malware. These challenges motivate Cubismo's design.

Challenge 1: Multiple Layers of Obfuscation and Dynamic Constructs. PHP malware heavily leverages obfuscation techniques, often using multiple layers of obfuscations to thwart static analysis's ability to recognize malicious code. As a result, many malware detectors decide to *simply flag obfuscated files as malware*. However, this strategy suffers from false positives as there are benign applications that use obfuscation techniques to protect intellectual property (e.g., source code or key algorithms) [29, 62].

PHP also supports various dynamic constructs that can create and modify program code introspectively (e.g., `Reflection`, `eval()` or `include()`). These dynamic constructs are commonly used in malware to implement obfuscation techniques. Moreover, PHP also provides methods to alter the current running program code, such as modifying and removing methods at runtime. This self-modification allows malware to delete or alter the critical code under analysis. Furthermore, PHP allows indirect and dynamic calling of functions (or methods) via a string variable that holds their name. Such features make it further challenging to statically reason about the code.

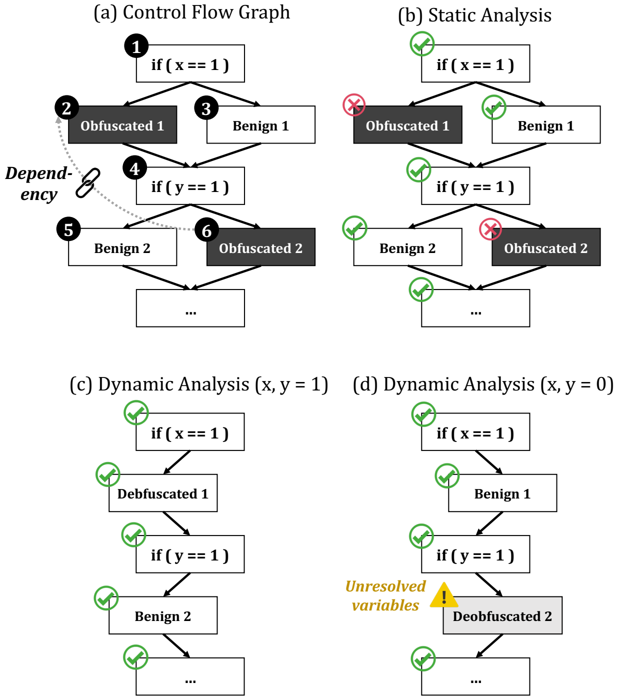
In short, static analysis based techniques are ineffective in analyzing malicious code hidden behind obfuscation. Note that ❷ and ❻ in Fig. 1-(a) are obfuscated. Static analysis techniques fail to see malicious code through the obfuscated blocks (depicted as ⊗ marks) in Fig. 1-(b).

To solve this problem, dynamic analysis based approaches are proposed. These approaches execute programs and observe their behavior to determine whether a target program is malicious. However, they are commonly thwarted by evasive logic in malware (details in challenge 2 below).

Challenge 2: Evasive Logic in Malware. Malware (including non-PHP malware) commonly leverages evasive techniques to hide malicious behaviors. Specifically, they often check host environment variables and client versions to identify which vulnerability they want to exploit [54]. In Fig. 1-(a), there are two obfuscated basic blocks (❷ and ❻) and to exercise both of them within an execution, the execution should satisfy the condition `x = 1` (❶) and `y != 1` (❹). In addition, note that the obfuscated block 2 (❻) is dependent on the obfuscated block 1 (❷), meaning that without a proper execution of the block 1, the block 2 would not execute as intended. While resolving the satisfying condition is trivial in this example, conditions can be arbitrarily complex in practice, making analysis challenging. As a result, simple dynamic analysis is often unable to completely expose hidden malicious code. Fig. 1-(c) and

[1] Cubismo is a Spanish word for cubism. This paper is inspired by cubist art which analyzes multiple aspects of an object, breaks them down, and reassembles them for presentation.

Figure 1: Limitations of Static and Dynamic Analysis


(a) Control Flow Graph (b) Static Analysis

(c) Dynamic Analysis (x, y = 1) (d) Dynamic Analysis (x, y = 0)

(d) illustrate that *dynamic analysis techniques can only see the program from a specific execution path, missing potentially malicious code.*

Moreover, executing code blocks does not always lead to successful analysis. For example, in Fig. 1, block 6's execution is dependent upon code executed in block 2, meaning that if block 2 is emitted from a dynamic execution, block 6 will not expose any malicious behavior (denoted with the yellow exclamation).

Challenge 3: Automated Tools for Malware Creators. Another challenge in detecting web server malware is that there are many automated tools used by attackers to create and obfuscate malware [40, 55]. In practice, attackers can easily generate a number of malware variants automatically with diverse obfuscations within the origin malware itself. Detectors often rely on known signatures of malware and heuristics that detect keywords and patterns of malicious code which can be easily changed by applying a simple obfuscation. As even a very small change (e.g., changing a variable/function name) in malware results in a different signature, such detectors have fundamental difficulty in catching up with the growing number of new malware variants. Note that our evaluation results also echo the prevalence of such malware variants. Specifically, we find many malware variants in the wild, are generated from a few original malware samples (§ 6.4).

Summary. Table 1 summarizes the effectiveness of different analysis techniques in handling the three challenges. We group existing malware detection techniques into three categories: signature-based, static analysis based, and dynamic analysis based. Signature-based techniques represent tools that utilize signatures (e.g., keywords and patterns in malware) to detect malware. Static analysis based techniques syntactically analyze malicious programs without running malware. Dynamic analysis based techniques run malware and monitor malicious behaviors exhibited at runtime.

	Obfuscation	Evasive Techs	Auto. Tools
Signature-based	×	△	×
Static Analysis	×	○	△
Dynamic Analysis	△	×	○

* ×: Ineffective, △: Partially effective, ○: Effective.

Table 1: Effectiveness of Malware Detection Techniques.

As shown in Table 1, signature-based techniques are not effective in handling obfuscation and dynamic constructs (Challenge 1) as well as malware variant generation tools (Challenge 3) because even a small change in malware variants may change its signature. Static analysis based techniques are effective in handling evasive techniques (Challenge 2) (e.g., complex predicates that mask malicious behaviors at runtime) while they have difficulty handling sophisticated obfuscations and identifying generated malware variants (Challenge 1 and 3). Dynamic analysis based techniques are effective in handling malware variant generation tools and obfuscation (Challenge 1 and 3) while they are often ineffective in analyzing evasive logic (Challenge 2).

3 SCOPE OF WORK

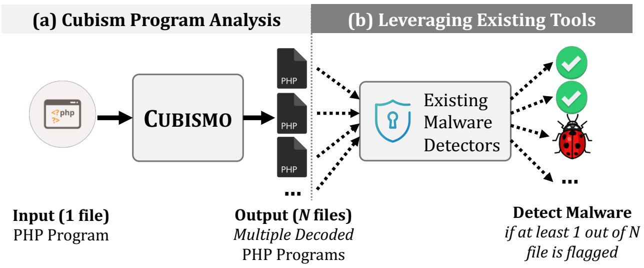
Cubismo is a system that reveals malicious code in malware and presents them as *multiple files* which can then be fed into existing malware detectors or analysis tools. It is important to mention that Cubismo itself is *not a malware detection technique* and it *does not decide* whether a given program is malware or not. Given an unknown malware, even after Cubismo reveals all the malicious behaviors, it is possible that all existing malware detection techniques are not able to detect the malware. In such case, it does not mean Cubismo is ineffective. It rather suggests that the existing detectors are unable to identify the given malware, indicating that it might be a completely new malware sample (i.e., zero-day malware).

Given a target program, Cubismo may generate multiple deobfuscated files if the target contains multiple layers of obfuscation or multiple instances of dynamic code generations (e.g., `eval()`). To confidently decide whether the given program is malicious or not, malware detectors need to scan *all of the generated files.* Hence, it often requires more time to scan all the files. Note that the goal of this paper is to create *a practical automated system that exposes malicious code of malware for existing malware detector tools*, not optimizing the detection pipeline. However, it should be obvious that one could prioritize and parallelize the scanning process to optimize performance.

4 OVERVIEW

Fig. 2 shows a typical workflow of Cubismo. Given an input PHP file, Cubismo exposes hidden program code blocks and merges each discovered hidden code block into the original input program to create a new output file. Note that it is important to merge deobfuscated code back to the original program because deobfuscated code alone may not form a complete malware. Given N revealed code blocks, Cubismo generates N output files. The output files are then fed into an existing malware detector such as VirusTotal [1]. If any of them are flagged as malware, the input file is likely malware.

Figure 2: Workflow of Cubismo


(a) Cubism Program Analysis
(b) Leveraging Existing Tools
php
Cubismo
PHP
PHP
PHP
...
Existing Malware Detectors
...
Input (1 file)
PHP Program
Output (N files)
Multiple Decoded PHP Programs
Detect Malware
if at least 1 out of N file is flagged

5 DESIGN

Fig. 3 shows the architecture of Cubismo. It consists of three phases: normalization (§ 5.1), counterfactual execution (§ 5.2), and code generator (§ 5.3).

Phase 1: Normalization. Cubismo statically parses the given input file to obtain an Abstract Syntax Tree (AST) of the input program. The AST is then normalized (e.g., pruning deprecated functionality, removing syntactically invalid statements). This normalized AST is then used in subsequent analyses instead of the original file, and is the output of this phase.

Phase 2: Counterfactual execution. Cubismo dynamically executes the input program under a controlled environment (i.e., a sandbox) to reveal malicious code. To enhance the code coverage that is a fundamental limitation of dynamic analysis, Cubismo leverages a dynamic analysis technique called *counterfactual execution* that drives executions into all observable control paths regardless of the predicate conditions (i.e., without solving the predicate conditions) to discover hidden malicious code and obtain decloaked ASTs.

Phase 3: Code generator. Cubismo creates multiple output program files from decloaked ASTs. The output files can then be fed into existing malware detection (or analysis) tools.

5.1 Normalization

Cubismo first parses an input program and obtains Abstract Syntax Tree (AST) of the program. During this process, certain parts of the code that do not affect the semantics such as deprecated code, syntactically invalid statements, extra whitespaces, and comments are pruned out.

Cubismo creates a one-to-one mapping between AST and code (which is useful when decloaking parts of the code) so that decloaked code can be properly merged back to the original AST to create multiple decloaked ASTs.

Fig. 4 shows an example of the normalization process. Given a difficult-to-read original program (Fig. 4-(a)), Cubismo parses it into an AST which is then used to create a normalized program (Fig. 4-(b)). Note that each statement is now on a separate line and some of PHP specific tags such as the closing tag ("`?>`") are removed, as they do not change the semantics of the original program in modern PHP syntax.

Based on our evaluations, we noticed that the normalization process in itself sometimes enhances detection results of existing detectors. Enhanced detect can occur because some malware detectors use subsequences of malware source code as a signature and some malware variants intentionally insert unnecessary code snippets (e.g., adding comments and closing tags) to break these signatures.

Malicious code is typically ill-formatted. More often than not, malicious code is injected in the middle of a benign file by the attackers, in a single line of code that constitutes several statements, using old PHP features to ensure portability. Moreover, *maliciously crafted statements may break specific PHP parsers* while they can be properly executed. For example, Fig. 5-(a) is a PHP program that contains *namespaces*. Note that it contains a line of comment outside of the namespaces (Line 13). The program runs correctly without any errors. However, PHP-Parser [44], a widely used parser for PHP programs, fails to parse the program, resulting in an error [16]. The normalization process removes the comment as shown in Fig. 5-(b), allowing the parser to parse the program without errors.

5.2 Counterfactual Execution

To expose hidden malicious code effectively, Cubismo systematically explores multiple execution paths of the target program to handle evasive techniques and obfuscation. Traditionally, exploring various execution paths requires either knowing various inputs to drive the execution paths or applying symbolic execution to resolve the predicate conditions [15]. However, due to, in part, the dynamic nature of PHP language and the sheer complexity of modern PHP programs including malware (e.g., reliance on several external applications and services in control-flow decisions), it is challenging to identify sufficient inputs or resolve predicate conditions via symbolic execution.

Instead, we leverage a concept called *counterfactual execution* which systematically explores *all executable statements*. It neither requires any inputs of the program, nor is based on symbolic execution which has difficulty handling a large number of complex conditions in real-world PHP programs.

Counterfactual execution enables discovery of parts of code that would not be accessible in a vanilla dynamic analysis [48]. Specifically, counterfactual execution (1) forcibly drives an execution into branches even if the branch conditions are not satisfied, (2) past exit nodes so that it can execute the pieces of code that are not normally covered, and (3) continuing executions when exceptions occur. It enables us to *unwrap, decode, and expose* the original code of obfuscated and encoded code.

Malicious Code Discovery in Dynamic Languages. Counterfactual execution shares the idea of forcing executions into all possible branches with multi-path exploration [36] and forced execution [34]. However, counterfactual execution differs from them in that it focuses on discovery of new code in addition to exploring all possible paths. Specifically, counterfactual execution treats dynamic constructs such as `eval()`, `include()`, and dynamic function calls, each of which might lead to discovery of new files and generation of new paths along the program execution, specially. It does so by creating nested isolated program states every time a branch or dynamic construct is encountered. The isolated states not only

Figure 3: Architecture of Cubismo

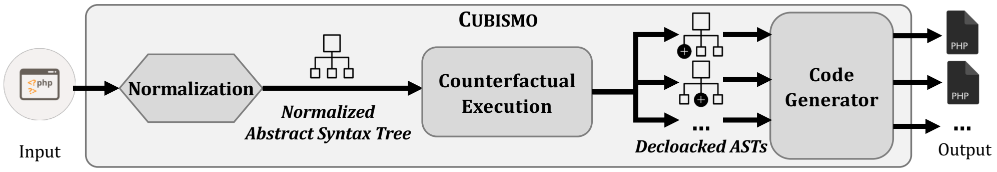
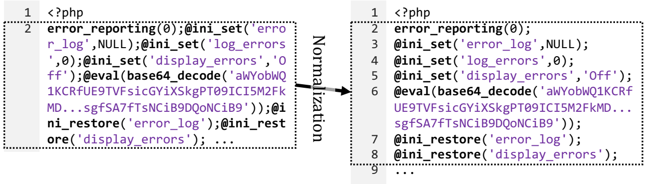


Figure 4: Example of Normalization Process


Normalization

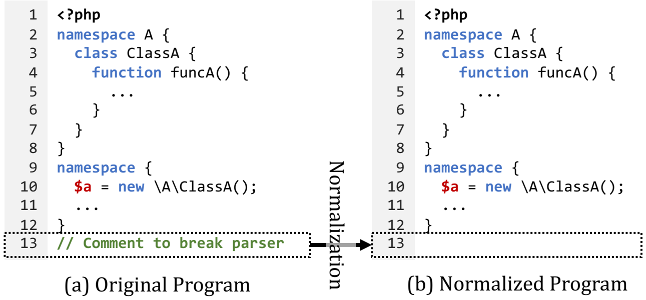
```
<?php
error_reporting(0);@ini_set('erro
r_log',NULL);@ini_set('log_errors
',0);@ini_set('display_errors','O
ff');@eval(base64_decode('aWYobWQ
1KCRfUE9TVFsicGYiXSkgPT09ICI5M2Fk
MD...sgfSA7fTsNCiB9DQoNCiB9'));@i
ni_restore('error_log');@ini_rest
ore('display_errors'); ...
```

(a) Original Program

```
<?php
error_reporting(0);
@ini_set('error_log',NULL);
@ini_set('log_errors',0);
@ini_set('display_errors','Off');
@eval(base64_decode('aWYobWQ1KCRf
UE9TVFsicGYiXSkgPT09ICI5M2FkMD...
sgfSA7fTsNCiB9DQoNCiB9'));
@ini_restore('error_log');
@ini_restore('display_errors');
...
```

(b) Normalized Program

Figure 5: Crash Inducing Program for PHP-Parser


Normalization

```
<?php
namespace A {
  class ClassA {
    function funcA() {
      ...
    }
  }
}
namespace {
  $a = new \A\ClassA();
  ...
}
// Comment to break parser
```

(a) Original Program
(PHP-Parser Crash)

```
<?php
namespace A {
  class ClassA {
    function funcA() {
      ...
    }
  }
}
namespace {
  $a = new \A\ClassA();
  ...
}

```

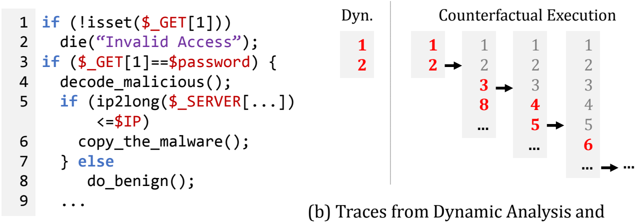
(b) Normalized Program
(No Crash)

ensure integrity of program state, they also enable continuation of execution past exceptions and fatal errors (by creating new isolations with negated error conditions). Moreover, PHP malware often uses nested predicates with dynamic constructs, making analysis attempts by symbolic execution approaches particularly challenging. Specifically, resolving string arguments of dynamic constructs (e.g., `eval()`) is a challenging problem in symbolic approaches. Counterfactual execution does not have such problems as it forcibly drives execution paths regardless of predicate conditions and handles runtime faults that can be caused by the forced execution paths via its sandboxed fail-free environment. When there is a runtime error or an exception that may terminate the execution, Cubismo creates a new nested isolated program state and continues the execution. Loops and recursive calls can also hinder analysis. We handle them by limiting the number of iterations and recursions (e.g., 100 iterations/recursion in this paper). For example, if a loop iterates more than 100 times, we terminate the loop by manipulating its control flow.

Sharing Artifacts between Isolated Executions. A vital feature of counterfactual execution is that it enables *sharing of discovered artifacts throughout isolated executions*. Database connections, file pointers, network connections, function and class redefinitions and system resources are among the artifacts that can be procured and shared via counterfactual execution. The importance of this *artifact sharing* is shown in Fig. 1, where the obfuscated code in block ❻ requires a key obtained in block ❷ to decloak itself. Without counterfactual execution forcing itself into block ❷ to discover and share the key, counterfactual execution of block ❻ will yield no malicious code.

When counterfactual execution encounters a dynamic construct statement, it checks if other isolated executions have different definitions for the statement (e.g., including different files via `include` or creating different dynamic code). If they do, counterfactual execution creates new isolated execution for each of the definitions from other isolated executions using the current execution context. We find that such new executions contribute to the discovery of new statements, resulting in the discovery of a significant proportion of the program code. Indeed our evaluation show that vanilla multi-path exploration (without sharing analysis results between explored paths) discovers 36,034 statements in Wordpress (a popular PHP application), whereas counterfactual execution discovers 58,786 statements, 63% more code.

Figure 6: Example of Counterfactual Execution Discovering Malicious Code



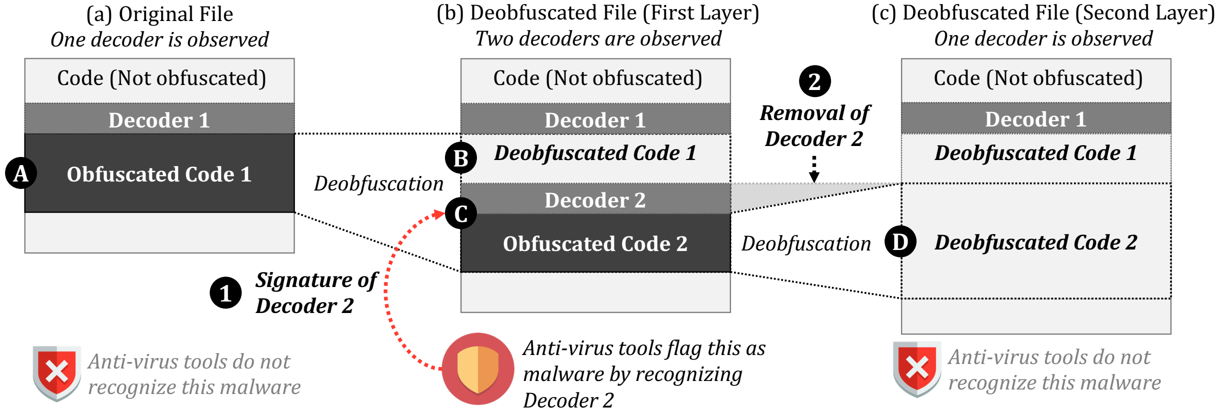
```
1 if (!isset($_GET[1]))
2   die(“Invalid Access”);
3 if ($_GET[1]==$password) {
4   decode_malicious();
5   if (ip2long($_SERVER[...])
        <=$IP)
6     copy_the_malware();
7   } else
8      do_benign();
9   ...
```

(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 state)

Robust Sandbox. Counterfactual execution is also bundled with a robust sandbox to prevent malicious behavior from affecting the host system, while guarding against reflective and introspective behavior. Counterfactual execution sandboxes more than 50 functions and classes of the original interpreter to make it ever harder for the malware to recognize it is being analyzed by Cubismo. Counterfactual execution analyzes dynamically generated code *recursively* until it comprehensively covers all possible dynamically generated code. PHP malware actively leverages recursive dynamic code generation.

Figure 7: Decloaking Process Example



Running Example. Consider the code in Fig. 6-(a). Line 1 checks if an input is provided to the script. When no input is available, line 2 exits the script (`die()` is the exit expression in PHP). Line 3 checks if the provided input is the expected password. If not, it executes benign statements on line 8 and exits. If the password is provided correctly, it decodes and executes malicious code (e.g., sending spam mails) on line 4. Then, it checks the range of client's IP address. If it is lower than a certain IP (e.g., `$IP` on line 5), it copies the malware which is another malicious activity (controlled spreading of the malware).

A naive dynamic analysis is unable to expose the malicious behavior as it is not able to drive execution past lines 2 and 3 (Dyn. in Fig. 6-(b)), missing the entire malicious logic.

5.3 Code Generator

For each decloaked AST, we generate a program file by traversing the tree. The generated program is different from the original program in two ways. First, it is based on the normalized AST from the first phase. The generated file may have different syntactic features such as indentations, whitespaces and line count. Second, it contains the decloaked version of obfuscated code in the original file, i.e., the respective parts of the AST that deobfuscate and execute dynamic code are replaced by the actual executed dynamic code.

Decloaking Process Example. Fig. 7 shows how Cubismo reveals malicious code in malware. First, Cubismo first normalizes the original input program and obtains its AST. Then, Cubismo uses the counterfactual execution to expose malicious code pieces. Specifically, whenever Cubismo executes dynamic constructs (e.g., Decoder 1 (🅐) in Fig. 7-(a)) that are used for deobfuscation, it replaces the resulting deobfuscated code (🅑 and 🅒) with the original (obfuscated) code as depicted in Fig. 7-(b). Malware may also include multiple layers of obfuscations. For instance, Fig. 7-(b)'s deobfuscated code includes another piece of nested obfuscated code (Obfuscated Code 2) and its decoder (Decoder 2) (🅒).

Cubismo repeatedly executes dynamic constructs (i.e., deobfuscator) until it does not observe any new resolvable dynamic construct. For instance, it executes the decoder 2 (🅒) to get the deobfuscated code 2 in Fig. 7-(c) and stops there as there is no more decoders that can expand code. Note that even if there is the decoder 1, executing it does not lead to newly observable code. Cubismo stops its exploration when arguments of dynamic constructs are attacker controlled inputs (e.g., `eval($_GET[$var])`). As a program containing dynamic constructs with inputs from untrusted sources (e.g., other websites) is generally considered malicious (e.g., web shell malware), Cubismo intentionally leaves such dynamic execution code intact so that detectors can use them to detect malware. For example, a malware detector that is aware of Decoder 2 (i.e., having a signature of Decoder 2) can detect the program after the first obfuscation layer (annotated with ❶) while it cannot detect the program after the second obfuscation layers (Fig. 7-(c)) as Decoder 2 is removed after its execution (annotated with ❷).

A malicious program may also alter itself in order to hinder analysis tools. For instance, after the deobfuscation in Fig. 7-(b), *Decoder 2 is removed* (❷) and is not a part of the program any more in Fig. 7-(c) (🅓). In PHP, alternation can be done by using the built-in tokenizer and code inspection and modification functions, which can remove an existing function in the current program. Because of these self-modifying behaviors, *the last deobfuscated file does not always contain all malicious code snippets.* It is also noteworthy that the obfuscations can be nested, and by unwrapping one layer of obfuscation, Cubismo can observe a new obfuscation in the generated code. There can be an arbitrary number of obfuscation layers.

6 EVALUATION

We evaluated Cubismo in order to answer the following research questions.

RQ 1. How effective is Cubismo in revealing malware disguised behind multiple layers of obfuscation? (§ 6.1)

RQ 2. Does Cubismo cause false positives on benign files? (§ 6.2)

RQ 3. What is the performance overhead? (§ 6.3)

RQ 4. How effective is Cubismo in handling real-world malware? (§ 6.4)

Experiment Setup. All experiments are run on an iMac 27" 2017 base model, running macOS 10.14.1 and PHP version 7.2. Samples were submitted to VirusTotal (VT) [1] via its API. Submissions are done in parallel and submitted samples are processed within a small time window (i.e., in a few minutes), reducing the side effects of our submission to the VT. As for each malware, the submissions are analyzed within a few minutes, we did not observe anti-virus engines in VT changing their behaviors because of our submissions (i.e., learn new malware samples because of our submission). From our experience, such learning behaviors happens a few days (e.g., 3-5 days) after the first submission. For example, many VT anti-virus engines were able to detect malware samples a few days after our submissions that they were not able to detect initially, .

Dataset Selection. To evaluate the effectiveness of Cubismo in practice, we leverage a large data set of real-world websites deployed in the wild obtained from a web hosting company, that maintains nightly backups of over 400,000 websites [2]. For each backup, Linux Malware Detector [47] is used to scan every file in the backup. Any website included in our dataset had at least one file flagged as malware. Hence, the dataset includes both benign and malicious PHP files, some of which are flagged by Linux Malware Detector. The total size of the dataset is 1TB including more than 3 million files. We filtered non-program files which are not the focus of this work, by parsing every file in the dataset and looking for PHP code in the parse tree, resulting in approximately 700k files.

From the 700k files, we selected files including dynamic code execution constructs (such as `eval()`, `create_function()`, `include()`, etc.), totaling 1,269 files with dynamic constructs. Note that this selection was static. We selected files that had dynamic code generation and execution constructs in their parse tree. We realized that a static filtering may not precisely identify all files containing dynamic features. However, we did this filtering to obtain a reasonable data set that includes PHP malware, and it was not meant to be exhaustive.

From the 1,269 files, we removed duplicate files, resulting in 1,040 unique files. These files were then all submitted to VT to get a baseline for detection. All but 352 (i.e., 688) files were detected by VT as previously known malware.

From the undetected 352 files, we manually inspected all files to obtain 56 malicious files. The *56* unique malicious files are zero-day PHP malware that are not detected by VT, and are the basis of this evaluation. We also manually selected 100 benign files from the remaining files for false positive evaluations (§ 6.2).

6.1 Decloaking Real-world Malware

By processing the 56 malware samples through Cubismo, we obtained 200 decloaked sample files that can be scanned by malware detection tools. We used VirusTotal (VT) as our malware detection oracle in this work, even though there are some other PHP malware detection tools [20, 43, 47]. VirusTotal is an aggregate virus scanning engine that scans submitted files with up to 60 different anti-virus engines, and aggregates the detection results.

The 200 files consisted of 56 original malware samples (which were undetected by VT), 56 normalized versions of the same malware samples, and 1 to 4 additional decloaked files per malware sample (depending on how many layers of obfuscation could be decloaked). All of these files were submitted to VT for scanning, and Table 2 shows the results. Each cell in the table shows how many engines in VT detected a particular file as malicious. Note that there are malware samples employing multiple obfuscation layers (namely `m2`, `m34`, `m40`, `m45`) to hide malicious behavior.

Observations. First, normalization is necessary in detecting real-world malware samples that actively exploit ill-formatted code snippets. Specifically, several malware samples (namely `m1`, `m10`, `m14`, `m19`, `m33`, `m36`, `m38`, `m42` and `m53`) are detected after normalization, even before decloaking. Our further investigations revealed that this is due to the fact that normalization fixes several dubious syntax issues present in the original malware.

Second, scanning each deobfuscated layer is necessary for accurate detection. Specifically, in some malware samples such as `m21`, we saw that decloaking the last obfuscation layer enables several more engines to detect maliciousness, jumping from 1 detection at first decloaking layer to 5 in the second layer. In other samples such as `m34`, we observed that the third decloaking layer results in no more engines discovering the maliciousness. This result is because the engines that detect malicious behavior on first and second decloaking layers have signatures for detecting obfuscation, which are removed from the code gradually. However, on the last (fourth) deobfuscation, a new engine recognizes the (now revealed) maliciousness. The same pattern can be observed on `m40` where detection goes from 3 to 1, and then back to 3. Complementary to this behavior, samples such as `m11`, `m14` and `m33` go from 3 detections on the first deobfuscation layer to 1 on the second layer. These patterns establish that *no particular layer is always the most suitable for detection*, and best result is achieved by scanning results of *all decloaking layers*.

Third, there are 3 samples that are not flagged as malware even after the decloaking: `m17`, `m27` and `m50`. These samples are true zero-days, i.e., no signature exists for them, and thus even after decloaking, the malicious behavior is not detected.

6.2 Cubismo on Benign Applications

To evaluate whether Cubismo introduces false positives in detection, as previously noted, we manually selected 100 benign PHP files from our dataset that included dynamic code generation and execution. Additionally, we randomly selected 200 files from Wordpress to evaluate whether Cubismo causes false positives in benign applications. These samples were decloaked using Cubismo and submitted to VT, along with their normalized and original versions. None of these samples and their different versions were flagged as malicious by VT (i.e., no false positives). Note that, for benign applications, Cubismo simply unfolds the dynamic constructs if possible, without changing any semantics. In benign applications, dynamically generated code should also be benign (e.g., they should not form webshells). Hence, we argue that doing so is unlikely to cause false positives (FP). Even if it does cause false positives, it means that malware detectors contain wrong signatures.

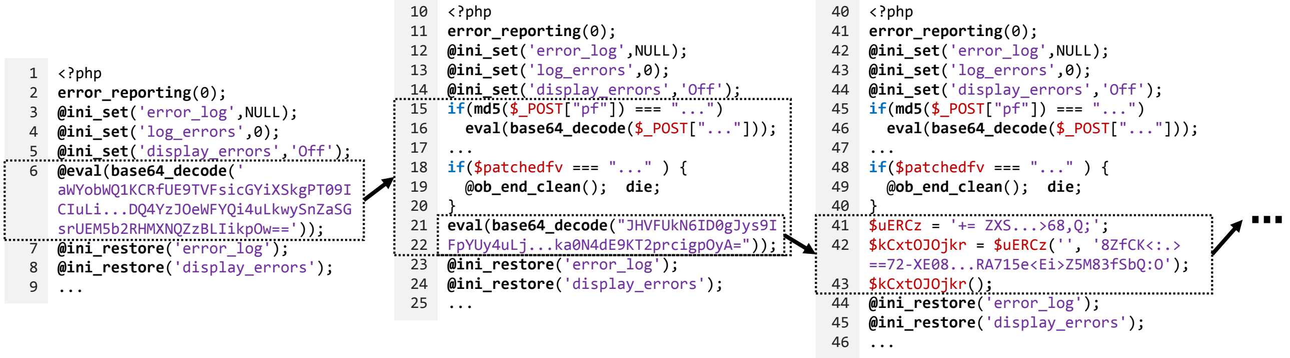
6.3 Performance

The execution times on Table 3 show that counterfactual execution typically takes around 20 milliseconds (ms) per sample. Some samples need up to 740 ms to completely decloak themselves. The average execution time is 52 ms. Scan times vary among different underlying scan engines. Antivirus engines typically take 10 to 120

Table 2: Results of Scanning Decloaked Malware Samples with VirusTotal

	Orig.	Norm.	Layer 1	Layer 2	Layer 3	Layer 4		Orig.	Norm.	Layer 1	Layer 2	Layer 3	Layer 4
m1	0	1	2	–	–	–	**m29**	0	0	1	1	1	–
m2	0	0	1	1	1	2	**m30**	0	0	1	–	–	–
m3	0	0	1	–	–	–	**m31**	0	0	1	–	–	–
m4	0	0	1	–	–	–	**m32**	0	0	1	–	–	–
m5	0	0	1	1	–	–	**m33**	0	1	3	1	–	–
m6	0	0	1	–	–	–	**m34**	0	0	1	1	0	1
m7	0	0	1	1	–	–	**m35**	0	0	1	–	–	–
m8	0	0	1	–	–	–	**m36**	0	1	1	–	–	–
m9	0	0	1	–	–	–	**m37**	0	0	1	–	–	–
m10	0	1	1	–	–	–	**m38**	0	1	1	–	–	–
m11	0	0	3	1	–	–	**m39**	0	0	1	–	–	–
m12	0	0	1	–	–	–	**m40**	0	0	3	1	3	3
m13	0	0	1	–	–	–	**m41**	0	1	1	–	–	–
m14	0	1	3	1	–	–	**m42**	0	1	1	–	–	–
m15	0	0	1	–	–	–	**m43**	0	0	1	1	1	–
m16	0	0	1	1	–	–	**m44**	0	0	1	1	–	–
m17	0	0	0	0	–	–	**m45**	0	0	1	1	1	1
m18	0	0	1	–	–	–	**m46**	0	0	1	–	–	–
m19	0	1	1	–	–	–	**m47**	0	0	1	–	–	–
m20	0	0	1	1	1	–	**m48**	0	0	1	–	–	–
m21	0	0	1	5	–	–	**m49**	0	0	1	1	–	–
m22	0	0	2	2	–	–	**m50**	0	0	0	–	–	–
m23	0	0	1	–	–	–	**m51**	0	0	1	–	–	–
m24	0	0	1	–	–	–	**m52**	0	0	1	–	–	–
m25	0	0	3	–	–	–	**m53**	0	1	2	1	–	–
m26	0	0	1	1	–	–	**m54**	0	0	1	–	–	–
m27	0	0	0	0	–	–	**m55**	0	0	1	–	–	–
m28	0	0	1	–	–	–	**m56**	0	0	1	–	–	–

Figure 8: Malware Sample with Multiple Layers of Obfuscations



```
 1  <?php
 2  error_reporting(0);
 3  @ini_set('error_log',NULL);
 4  @ini_set('log_errors',0);
 5  @ini_set('display_errors','Off');
 6  @eval(base64_decode('
    aWYobWQ1KCRfUE9TVFsicGYiXSkgPT09I
    CIuLi...DQ4YzJOeWFYQi4uLkwySnZaSG
    srUEM5b2RHMXNQZzBLIikpOw=='));
 7  @ini_restore('error_log');
 8  @ini_restore('display_errors');
 9  ...
```

(a) Normalized Program

```
10  <?php
11  error_reporting(0);
12  @ini_set('error_log',NULL);
13  @ini_set('log_errors',0);
14  @ini_set('display_errors','Off');
15  if(md5($_POST["pf"]) === "...")
16    eval(base64_decode($_POST["..."]));
17  ...
18  if($patchedfv === "..." ) {
19    @ob_end_clean();  die;
20  }
21  eval(base64_decode("JHVFUkN6ID0gJys9I
22  FpYUy4uLj...ka0N4dE9KT2prcigpOyA="));
23  @ini_restore('error_log');
24  @ini_restore('display_errors');
25  ...
```

(b) Deobfuscated Program 1

```
40  <?php
41  error_reporting(0);
42  @ini_set('error_log',NULL);
43  @ini_set('log_errors',0);
44  @ini_set('display_errors','Off');
45  if(md5($_POST["pf"]) === "...")
46    eval(base64_decode($_POST["..."]));
47  ...
48  if($patchedfv === "..." ) {
49    @ob_end_clean();  die;
40  }
41  $uERCz = '+= ZXS...>68,Q;';
42  $kCxtOJOjkr = $uERCz('', '8ZfCK<:.>
    ==72-XE08...RA715e<Ei>Z5M83fSbQ:O');
43  $kCxtOJOjkr();
44  @ini_restore('error_log');
45  @ini_restore('display_errors');
46  ...
```

(c) Deobfuscated Program 2

ms to scan each individual file. Another factor impacting performance is that each file is converted into multiple files, each of which need to be scanned by the underlying detection engine. With an average of 4 files per malware sample, underlying engine scan times can take from 40 to 480 ms, on average. The decloaking process would then be adding 11% to 130% runtime overhead. However, these times are only applicable for files that include dynamic code generation. For memory space overhead, counterfactual execution needs less than 200MB of memory.

6.4 Case Study

We dissect two real-world malware samples that Cubismo reveals their hidden malicious code to show the effectiveness.

6.4.1 *Exposing Multiple Layers of Malware.*

Fig. 8 shows a zero-day malware sample with multiple obfuscation layers. Fig. 8-(b) shows its first deobfuscated layer. Specifically, line 6 in Fig. 8-(a) is deobfuscated to lines 15-22 in Fig. 8-(b). Note that from a single line, multiple lines (i.e., lines 15-22) are generated. Then, Fig. 8-(c) represents another deobfuscated piece of code from Fig. 8-(b). Note that while Fig. 8-(b) has multiple instances of dynamic constructs (e.g., `eval()`) on lines 16 and 21, only the `eval()` on line 21 is deobfuscated. This obfuscation occurs because `eval()` on line 16 depends on an attacker provided input which cannot be resolved. Interestingly, we observed a unique obfuscation technique on lines 41-43 in Fig. 8-(c). When executed, the malware generates another `eval()` function with `base64_encoding`. We omit the details about subsequent obfuscation layers due to space constraints.

Analysis. We scanned each obfuscated program with VirusTotal (VT). Note that submissions of the deobfuscated files are done in parallel within a small time window (i.e., in a few seconds). The deobfuscated program 1 (Fig. 8-(b)) is flagged as malware by one tool called `bkav` [10], showing that resolving obfuscation layers via Cubismo is effective in practice.

The deobfuscated program 2 (Fig. 8-(c)), however, is not detected as malware due to the new code generated from the previous program (Fig. 8-(b)), resulting in a different signature. Note that the deobfuscated program 2 is not flagged *while it includes malicious code from the deobfuscated program 1* which was already detected as malware. This result essentially shows that the detection accuracy of the underlying tools is very sensitive to changes in the target program, suggesting a fundamental limitation of signature based techniques.

Malware detectors detect the following layers' deobfuscated code after the deobfuscated program 2 as malware. It also shows that on each obfuscated layer, different malicious code and obfuscation techniques can be used. As malware detectors may only recognize some of those techniques, analyzing individual deobfuscated layer independently can increase the chance to detect malware compared to analyzing the last deobfuscated layer. Note that the deobfuscation process may remove or alter the program itself. Hence, the last deobfuscated file may not include all the malicious code.

6.4.2 *Handling Variants of Existing Malware.*

Fig. 9-(a) shows the one malware sample which has many polymorphic variants throughout the data set. The code decompresses a string that represents a zip stream returned from the `base64_decode` function on line 2. Then, the `eval` function on line 3 dynamically runs the decompressed string. The size of the original malware sample is 34 KB, and is simplified in the figure.

Note that an automated tool can easily be used to create the variants shown in Fig. 9-(b), (c), and (d). The variants are semantically identical. Each variant has only two polymorphic differences: (1) the value of `$s_pass` (lines 10, 20, and 30) which is essentially a hashed password used by the malware, and (2) the means of splitting the string used in `eval()` (lines 11, 21, and 31). It is trivial to create a new password hash and to split a string into multiple substrings with different lengths, which are essentially the only two changes used to automate this process.

Detecting Malware via Cubismo. Fig. 9-(e) shows the deobfuscated malware from the variants decloaked through Cubismo. The malware contains HTML tags on lines 41-42 (in `$buff`) as well as lines 45-48. The malware is a popular webshell called `b374k` [5], which provides ssh-like access to the web server via a web interface.

We used VT to scan the original malware and 10 different variants generated in the same fashion as shown in Fig. 9-(b), (c), and (d). Only one engine (`bkav` [10]) detected the original malware. Consequently, none of the 60 malware detectors in VT were able to detect any of the polymorphic variants, showing the effectiveness effective of a simple polymorphic malware variant generation technique. The deobfuscated result of the malware obtained via Cubismo, Fig. 9-(e), is detected by 4 engines of VT (`Avast` [3], `AVG` [4], `Baidu` [6], and `bkav`), showing Cubismo's effectiveness in detecting variants.

7 DISCUSSION

Signature Updates of AV Tools. Once we submit malware samples to VT, it is possible that AV engines obtain the samples and analyze them to update their signatures. In fact, we have observed such updates two weeks after our submission. As we used automated scripts to submit all samples, during our experiments no such issues arose. After two weeks, we noticed that several samples (`m2`, `m17`, `m27`, `m33`, `m40`, and `m45`) are now detected by VT in their original state. Furthermore, `m17`, `m27`, and `m50`, which remained undetected even after decloaking in the original experiments, were still undetected. This result is most likely because the signatures defined for these new malware samples are not mature yet, and narrowly match the original file, hence any modifications to the original file will result in no detection.

Hiding Malicious Code Snippets in Comments. It is possible for malware to store its payload in comments that are removed during normalization (as a means of bypassing Cubismo). The current implementation of Cubismo is not able to handle such malware. However, we did not observe any such cases in our evaluations. We leave handling such malware to future work.

Overhead When Integrated into Malware Detectors. When Cubismo is integrated into existing malware detectors to provide an end-to-end malware detector system, additional overhead can be introduced because Cubismo generates multiple decloaked files (4 files on average) from a single malware sample to be scanned. To mitigate this additional overhead, we can run malware detectors in parallel.

Specifically, if we assume that malware detectors' executions can be fully parallelized without additional overhead, Cubismo will cause additional 52ms overhead as reported in § 6.3. When we parallelize the scanning task, we create multiple malware detector processes by forking an already initialized process (i.e., a Zygote process) to minimize the overhead of process initialization. To this end, we minimize the overhead of parallelizing, resulting in less than 10% overhead (i.e., 5 ∼ 10 ms) on average when there are sufficient computing resources.

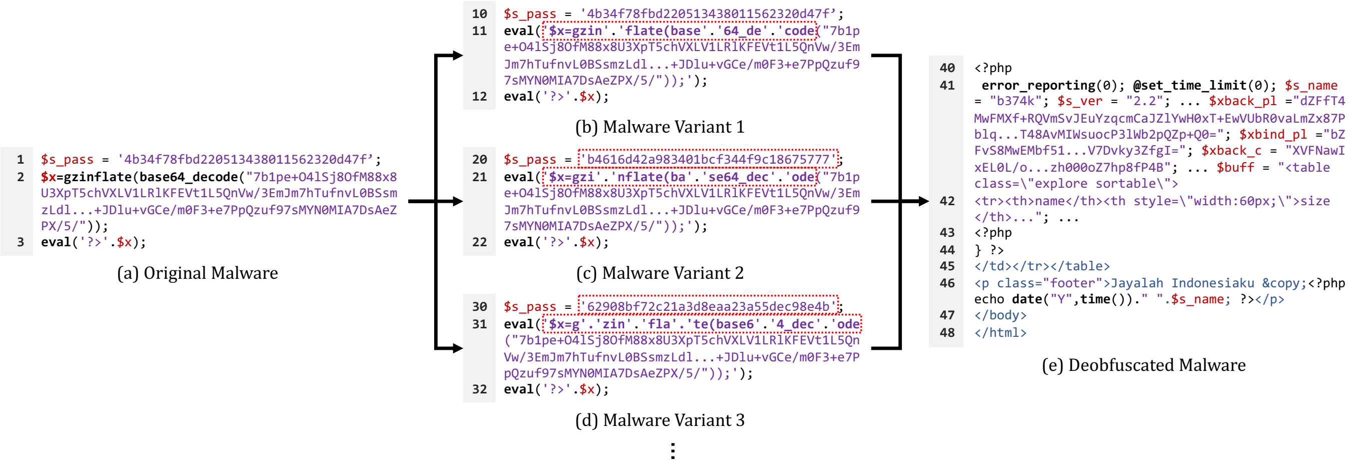
In the worst case scenario (i.e., when we cannot run malware detectors in parallel), Cubismo incurs approximately N times slowdown where N is the number of decloaked files (N is 4 in our experiments). However, we argue that most modern machines can run more than 4 instances of malware detectors at the same time without significant additional overhead. We leave improving the efficiency of Cubismo under the malware detector integration scenario as future work.

8 RELATED WORK

Research in dynamic server-side code analysis has opened the door for our work [25].

Malicious Payload/Behavior Discovery. A large number of related work have focused on discovering malicious payloads on servers by investigating their client-side HTML and JavaScript output [7, 12, 17, 28, 32, 50, 57]. However, they may not reveal the existence of evasive malware on a server reliably. Malware that can recognize detection attempts do not emit full behavior to the client [17]. There has been a line of research that focuses on discovering malicious behavior in binaries [8, 21, 35, 39], in contrast with dynamic scripted code, binaries require code that enables them to pervasively modify themselves. Many of these works focus on finding the self-modifying code generator, which has a distinct signature in binaries, rather than the actual malicious behavior, which is hard to pinpoint due to the diverse functionality of binaries [9, 19, 41].

Figure 9: Polymorphic Variants



```
$s_pass = '4b34f78fbd220513438011562320d47f’;
$x=gzinflate(base64_decode("7b1pe+O41Sj8OfM88x8U3XpT5chVXLV1LRlKFEVt1L5QnVw/3EmJm7hTufnvL0BSsmzLdl...+JDlu+vGCe/m0F3+e7PpQzuf97sMYN0MIA7DsAeZPX/5/"));
eval('?>'.$x);
```

(a) Original Malware

```
$s_pass = '4b34f78fbd220513438011562320d47f’;
eval('$x=gzin'.'flate(base'.'64_de'.'code("7b1pe+O41Sj8OfM88x8U3XpT5chVXLV1LRlKFEVt1L5QnVw/3EmJm7hTufnvL0BSsmzLdl...+JDlu+vGCe/m0F3+e7PpQzuf97sMYN0MIA7DsAeZPX/5/"));');
eval('?>'.$x);
```

(b) Malware Variant 1

```
$s_pass = 'b4616d42a983401bcf344f9c18675777';
eval('$x=gzi'.'nflate(ba'.'se64_dec'.'ode("7b1pe+O41Sj8OfM88x8U3XpT5chVXLV1LRlKFEVt1L5QnVw/3EmJm7hTufnvL0BSsmzLdl...+JDlu+vGCe/m0F3+e7PpQzuf97sMYN0MIA7DsAeZPX/5/"));');
eval('?>'.$x);
```

(c) Malware Variant 2

```
$s_pass = '62908bf72c21a3d8eaa23a55dec98e4b';
eval('$x=g'.'zin'.'fla'.'te(base6'.'4_dec'.'ode("7b1pe+O41Sj8OfM88x8U3XpT5chVXLV1LRlKFEVt1L5QnVw/3EmJm7hTufnvL0BSsmzLdl...+JDlu+vGCe/m0F3+e7PpQzuf97sMYN0MIA7DsAeZPX/5/"));');
eval('?>'.$x);
```

(d) Malware Variant 3

```
<?php
 error_reporting(0); @set_time_limit(0); $s_name = "b374k"; $s_ver = "2.2"; ... $xback_pl ="dZFfT4MwFMXf+RQVmSvJEuYzqcmCaJZlYwH0xT+EwVUbR0vaLmZx87Pblq...T48AvMIWsuocP3lWb2pQZp+Q0="; $xbind_pl ="bZFvS8MwEMbf51...V7Dvky3ZfgI="; $xback_c = "XVFNawIxEL0L/o...zh000oZ7hp8fP4B"; ... $buff = "<table class=\"explore sortable\">
<tr><th>name</th><th style=\"width:60px;\">size </th>..."; ...
<?php
} ?>
</td></tr></table>
<p class="footer">Jayalah Indonesiaku &copy;<?php echo date("Y",time())." ".$s_name; ?></p>
</body>
</html>
```

(e) Deobfuscated Malware

Counterfactual Execution. Counterfactual execution (or forced-execution) as a means of discovering the hidden behavior of malware has been used [13, 22, 34, 41, 45, 61]. In particular, Peng et. al. advance counterfactual execution on binaries by providing several error recovery features as well as better path expansion algorithms [45]. Unlike that work, Cubismo handles a new set of challenges caused by server-side dynamic languages such as dynamic constructs and multiple layers of obfuscations (Details in §2). J-Force [34] and Rozzle [36] use a similar method to analyze JavaScript malware (i.e., client-side malware). Specifically, Cubismo's underlying analysis technique, called counterfactual execution, shares the basic idea of forcing the execution into all possibilities. However, counterfactual execution differs from forced-execution [34] and multi-path exploration [36] as it shares analysis artifacts (e.g., database connections and file pointers) between isolated executions (Described in Section 5.2). Unlike client-side malware that J-Force and Rozzle target, server-side malware are often injected into large and complex benign application frameworks such as Wordpress and Joomla. To reveal injected malicious code snippets in such benign applications, simply exploring all branches is not sufficient as statements in the explored branches may depend on resources such as database connections and file handles created in other execution paths or functions. As a result, the artifact sharing scheme is crucial for revealing server-side malware.

Furthermore, Cubismo focuses on handling server-side specific evasive techniques such as obfuscations and polymorphic malware as discussed in §2. To name a few, Cubismo handles recursive dynamic code generations (e.g., recursive `eval`) and code generation across executions (e.g., metamorphic malware that generates new malware during its execution). Hallahan et al. [24] introduce a concept of *counterfactual symbolic execution*. While using a similar term, they explore differences between two alternative implementations of a function (e.g., the function's concrete implementation and an abstract implementation derived from the function's specification) whereas Cubismo explores all possible dynamic execution paths of a concrete program (i.e., a concrete implementation). Moreover, the goal of Cubismo differs significantly from [24] which aims to identify the causes of failure of static verifications (i.e., fault localization). We focus on exposing hidden malicious code in dynamic malware.

9 TOOL AVAILABILITY

To help facilitate future research, we are releasing the research artifacts, including the source code and datasets, to the research community [42].

10 CONCLUSION

In this paper, we describe the problem of obfuscated dynamic web server malware and its impact. We presented a practical system, Cubismo, that enables decloaking of highly evasive server-side malware. Cubismo is capable of generating decloaked versions of the original malware in which the obfuscated parts are replaced with deobfuscated code. It enables traditional malware detection engines such as VirusTotal to detect obfuscated malware. Our evaluations on real-world website data show that it enables detection of 53 out of 56 zero-day malware samples in the data set.

ACKNOWLEDGMENTS

We thank the anonymous referees and our shepherd Kevin Roundy 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] Anonymous. 2019. Anonymized-for-review. https://www.anonymous.com/. (2019).
[3] Avast Software. 2019. Avast Antivirus. https://www.avast.com/. (2019).
[4] Avast Software. 2019. AVG Antivirus. https://www.avg.com/. (2019).
[5] b374k. 2019. PHP Webshell with handy features. https://github.com/b374k/b374k. (2019).

[6] Baidu. 2019. Baidu Antivirus. http://sd.baidu.com/. (2019).

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

[8] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2010. Efficient detection of split personalities in malware. In *NDSS 2010, 17th Annual Network and Distributed System Security Symposium, February 28th-March 3rd, 2010, San Diego, USA*. San Diego, UNITED STATES. http://www.eurecom.fr/publication/3022

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

[10] Bkav Corporation. 2019. Bkav Internet Security. http://www.bkav.com/bkav-internet-security. (2019).

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

[12] Kevin Borgolte, Christopher Kruegel, and Giovanni Vigna. 2013. *Delta: automatic identification of unknown web-based infection campaigns*. ACM.

[13] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. 2008. *Automatically Identifying Trigger-based Behavior in Malware*. Springer US, Boston, MA, 65–88. https://doi.org/10.1007/978-0-387-68768-1_4

[14] 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

[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 265–278. https://doi.org/10.1145/1950365.1950396

[16] Christian Johansson. 2017. Doesn't support comments outside of namespaces declared with bracketed syntax. https://github.com/nikic/PHP-Parser/issues/412. (2017). Accessed: 2019-05-30.

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

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

[19] 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.

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

[21] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/1455770.1455779

[22] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, Berkeley, CA, USA, 303–317. http://dl.acm.org/citation.cfm?id=2671225.2671245

[23] 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).

[24] William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. 2019. Lazy Counterfactual Symbolic Execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 411–424. https://doi.org/10.1145/3314221.3314618

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

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

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

[29] ionCube Ltd. 2019. ionCube. https://www.ioncube.com/php$_e$ncoder.php. (2019).

[30] 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.

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

[32] 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).

[33] 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).

[34] 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.

[35] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. 2009. Effective and Efficient Malware Detection at the End Host. In *Proceedings of the 18th Conference on USENIX Security Symposium (SSYM'09)*. USENIX Association, Berkeley, CA, USA, 351–366. http://dl.acm.org/citation.cfm?id=1855768.1855790

[36] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2012. Rozzle: De-cloaking Internet Malware. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 443–457. https://doi.org/10.1109/SP.2012.48

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

[38] 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.

[39] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. 2011. Detecting Environment-sensitive Malware. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID'11)*. Springer-Verlag, Berlin, Heidelberg, 338–357. https://doi.org/10.1007/978-3-642-23644-$0_1$8

[40] mobilefish.coml. 2019. Simple online PHP obfuscator. https://www.mobilefish.com/services/php$_o$bfuscator/php$_o$bfuscator.php. (2019).

[41] 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.

[42] Naderi-Afooshteh, Abbas and Kwon, Yonghwi and Nguyen-Tuong, Anh and Bagheri-Marzijarani, Mandana and Davidson, Jack. 2019. CUBISMO Research Artifacts. https://cubismo.s3.amazonaws.com/cubismo.html. (2019).

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

[44] Nikita Popov. 2019. PHP-Parser. https://github.com/nikic/PHP-Parser. (2019). Accessed: 2019-05-30.

[45] 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).

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

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

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

[49] James Scott. 2017. Signature Based Malware Detection is Dead. (2017).

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

[51] 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.

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

[53] 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.

[54] Symantec. 2019. 2019 Internet Security Threat Report. https://www.symantec.com/security-center/threat-report. (2019).

[55] Vojtěch Sokol. 2019. srcProtector for PHP. http://phpobfuscator.net/. (2019).

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

[57] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghwi Kwon, Xiangyu Zhang, and Patrick Eugster. 2016. WebRanz: Web Page Randomization for Better Advertisement Delivery and Web-bot Prevention. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 205–216. https://doi.org/10.1145/2950290.2950352

[58] 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.

[59] 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.

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

[61] Zhaoyan Xu, Lingfeng Chen, Guofei Gu, and Christopher Kruegel. 2012. PeerPress: Utilizing Enemies' P2P Strength Against Them. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 581–592. https://doi.org/10.1145/2382196.2382257

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

A EFFECTIVENESS OF COUNTERFACTUAL EXECUTION

Table 3 lists details of counterfactual execution on the 56 malware samples, including the depth of nested dynamic evaluations, file size, number of statements per file, and number of statements generated dynamically, etc. On average, counterfactual execution is unveiling 45 new statements per sample (19% increase), and is exercising 9 new paths (48% increase) in the program.

Revelaing Hidden Malicious Code. Counterfactual execution is highly effective in revealing hidden malicious code snippets. In particular, `m21` has only 1 statement (highlighted in Table 3), which is a deobfuscation and dynamic code execution statement, resulting in generation of 475 new statements. This sample is a fully obfuscated malware, in contrast to many other samples, where an obfuscated malicious block is injected to a pre-existing, benign application file. `m48` is also another malware that is very similar, using 4 statements (highlighted) to deobfuscate and dynamically execute itself.

Undetected Zero-day Malware. `m17`, `m27`, and `m50` (highlighted rows) are zero-day malware samples that are not detected by existing malware detectors. As apparent in Table 3, `m17` and `m27` are from the same malware family (their analysis results are identical). This shows that malware variants are commonly observed in practice. `m50` is different from the other two, meaning that it might be originated from another malware family. Note that those are undetected not because of Cubismo, but because existing malware detectors do not have signatures for them. In fact, our analysis result provides useful information such as `m17` and `m27` are variants from the same malware family. It essentially suggests that deploying Cubismo would significantly reduce the manual efforts to handle variants. Specifically, with Cubismo, a single signature will be sufficient to detect two variants `m17` and `m27`.

Unresolvable Dynamic Constructs. Note the *Unresolvable Obfuscations* column in Table 3, which shows the number of dynamically generated code snippets that could not be resolved through counterfactual execution. This is mostly due to the fact that these blocks are directly generated by user input (i.e., user input gets executed as code). Such patterns are common in malware, and serve as an omnipotent backdoor to the application. It is vital that we do not attempt to decloak (i.e., replacing with nothing, as there is no user input) such dynamic executions, thus removing them from the file, as they are often used as the signature to detect backdoors and webshells.

Table 3: Details of Counterfactual Execution of Malware Samples

	Parsed Statements	Dynamic Executions	Dynamic Execution Nestings	Dynamic Statements Generated	Conditional Count	Branch Count	Counterfactual Conditional Count	Counterfactual Branch Count	Unresolvable Obfuscations	Execution Time	File Size
m1	138	3	2	17	10	13	4	7	2	20ms	10.4 KB
m2	1	7	2	57	30	36	7	13	3	740ms	573.2 KB
m3	62	3	2	17	10	13	4	7	2	20ms	18.4 KB
m4	66	3	2	17	10	13	4	7	2	10ms	11.8 KB
m5	40	4	2	22	14	17	4	7	2	20ms	10.2 KB
m6	70	3	2	17	10	13	4	7	2	20ms	18.4 KB
m7	194	4	2	22	16	19	4	7	2	20ms	15.2 KB
m8	2	3	2	17	10	13	4	7	2	10ms	3.5 KB
m9	42	3	2	17	10	13	4	7	2	10ms	5.9 KB
m10	210	3	2	17	10	13	4	7	2	30ms	11.7 KB
m11	8	4	2	22	14	17	4	7	2	10ms	4.0 KB
m12	434	3	2	17	11	14	4	7	2	50ms	30.4 KB
m13	254	3	2	17	10	13	4	7	2	40ms	17.3 KB
m14	40	4	2	22	14	17	4	7	2	20ms	7.0 KB
m15	32	3	2	17	10	13	4	7	2	10ms	4.7 KB
m16	1	4	2	22	14	17	4	7	2	20ms	6.1 KB
m17	1828	4	2	22	22	30	10	17	2	170ms	151.9 KB
m18	10	3	2	17	10	13	4	7	2	20ms	24.9 KB
m19	156	3	2	17	10	13	4	7	2	20ms	11.4 KB
m20	1	5	2	27	21	24	4	7	2	20ms	9.1 KB
m21	1	2	1	475	9	12	2	3	0	60ms	30.8 KB
m22	1	4	2	22	14	17	4	7	2	20ms	6.0 KB
m23	56	3	2	17	10	13	4	7	2	70ms	11.9 KB
m24	2	3	2	17	10	13	4	7	2	30ms	14.7 KB
m25	20	3	2	17	10	13	4	7	2	10ms	4.3 KB
m26	104	4	2	22	14	17	4	7	2	20ms	9.7 KB
m27	1828	4	2	22	22	30	10	17	2	180ms	151.9 KB
m28	40	3	2	17	10	13	4	7	2	20ms	11.8 KB
m29	214	6	2	39	23	29	7	13	3	30ms	25.2 KB
m30	718	3	2	17	10	13	4	7	2	80ms	75.1 KB
m31	58	3	2	17	10	13	4	7	2	20ms	15.9 KB
m32	276	3	2	17	10	13	4	7	2	60ms	38.5 KB
m33	52	4	2	22	14	17	4	7	2	20ms	4.8 KB
m34	1214	7	2	57	30	36	7	13	3	160ms	140.4 KB
m35	34	3	2	17	10	13	4	7	2	10ms	14.6 KB
m36	158	3	2	17	10	13	4	7	2	30ms	13.5 KB
m37	28	3	2	17	10	13	4	7	2	20ms	10.3 KB
m38	454	3	2	17	10	13	4	7	2	40ms	18.5 KB
m39	76	3	2	17	10	13	4	7	2	30ms	20.5 KB
m40	10	7	2	57	31	38	7	13	3	30ms	10.8 KB
m41	140	3	2	17	10	13	4	7	2	30ms	12.6 KB
m42	124	3	2	17	10	13	4	7	2	20ms	9.0 KB
m43	1	5	2	39	20	23	4	7	2	20ms	9.1 KB
m44	1	4	2	22	14	17	4	7	2	20ms	5.3 KB
m45	578	7	2	57	30	36	7	13	3	120ms	206.6 KB
m46	1104	3	2	17	10	13	4	7	2	80ms	119.7 KB
m47	148	3	2	17	10	13	4	7	2	30ms	51.5 KB
m48	4	2	2	885	49	59	44	53	1	150ms	19.7 KB
m49	1236	4	2	22	14	17	4	7	2	140ms	136.9 KB
m50	198	1	1	1	59	68	16	24	0	30ms	5.7 KB
m51	104	3	2	17	10	13	4	7	2	40ms	50.0 KB
m52	422	3	2	17	10	13	4	7	2	40ms	18.8 KB
m53	4	4	2	22	15	18	4	7	2	10ms	4.0 KB
m54	208	3	2	17	10	13	4	7	2	40ms	34.1 KB
m55	2	3	2	17	10	13	4	7	2	10ms	3.4 KB
m56	230	3	2	17	10	13	4	7	2	30ms	29.6 KB