MASTERMIND: Delivering Malicious Payloads via Innocent Actors

Abstract

Cybercriminals actively leverage various techniques to deliver malicious payloads and hinder forensic investigations. In this paper, we propose a new sophisticated server-side web attack, called MASTERMIND, that transforms contents from benign websites to malicious payloads through a reverse-engineering and forensic analysis resilient transformation technique. The core of the proposed technique is the fail-free state-machine that always makes a transition on any inputs. Moreover, outputs of the state-machine is dynamically changing depending on its inputs, making input and output spaces extremely large. We have developed a fully automated prototype, MINDGEN, that can generate MASTERMIND instances. Our evaluation results show that state-of-the-art analysis tools including static analysis tools, dynamic analysis tools, and symbolic execution tools were not able to detect, understand, and reveal malicious payloads hidden in MASTERMIND. Further, we demonstrate that, even for manual analysis, it is challenging to analyze MASTERMIND due to the design of our fail-free state-machine.

1 Introduction

Cyber attacks are becoming more and more sophisticated. In particular, recent advanced attacks are leveraging various complex methods to distract security analysts who want to understand real intentions and attackers behind the attacks. For example, in Advanced Persistent Threats, malware leverages benign and popular social media services (e.g., Twitter, Instagram, and YouTube) as Command and Control (C&C) servers [1] to avoid detection and hinder the effort to triage cybercriminals. Specifically, such malware connects to a benign webpage to receive secret messages that include either a secret command that can trigger malicious code in malware or executable malicious payloads. As the malware communicates only with benign network service providers such as Twitter, it can successfully avoid detection of existing defense techniques (e.g., firewalls and anti-virus solutions) that rely on blacklists or whitelists of known malicious IP addresses and domain names.

To understand such attacks, advanced malware analysis techniques have been proposed: (1) program analysis techniques [28,38,39,44,46,47,50,61,63,70–72,74,75,83,86,102,107] such as symbolic execution [19,32,84] and forced execution [46,47,75] have been used to uncover and analyze hidden malicious logic in malware and (2) deep packet inspection techniques [15,29,53,89] that can see through malicious payloads hiding in network packets. For malware that execute hidden malicious code triggered by a secret command, advanced program analysis techniques are highly effective in revealing the hidden code and secret command (i.e., attack-triggering inputs). On the other hand, for malware that receive malicious payloads from other servers and execute directly, deep packet inspection techniques [15,29,53,89] can be used to dissect network packets to discover malicious payloads encoded in the packets. Other evasive techniques such as obfuscations can be also detected and disarmed by state-of-the-art analysis tools or their combinations [9, 98, 106]. More importantly, we observe that existing attacks, while they are advanced and sophisticated, once they are disarmed and analyzed, it is *straightforward to triage forensically*. For example, even for the attacks that leverage benign services as C&C servers, malicious actors can be revealed by tracking down the accounts and network traces (e.g., IP logs) involved in the attacks.

In this paper, we explore the possibility of a new anti-post-mortem forensic attack, dubbed *Mastermind Attack*[1] (MASTERMIND), that evades state-of-the-art malware detection and analysis techniques and imposes fundamental challenges for forensic triage. It orchestrates benign websites (that are irrelevant to the attack) to collect unsuspicious inputs (e.g., non-executable code). Then, we use our proposed Fail-free dynamic state-machine (F^2DSM)[2] (Section 3.2) to transform the collected benign contents into a malicious payload, where the translation process is extremely challenging to reverse-engineer. Specifically, our novel fail-free state-machine allows state transitions for any inputs from any states, extending its input space. Moreover, outputs of the state-machine is dynamic, meaning that outputs are dependent on inputs of the state-machine, making both input and output space extremely large, hence *making it difficult to reverse-engineer*.

MASTERMIND does not include malicious code in itself as an executable form, hence analyzing our malware program code would not reveal real intentions of the attack. In addition, existing analysis techniques such as symbolic execution

[1] A mastermind is a person who supplies the directing or intelligence for a project. In particular, a criminal mastermind is the one who creates the blueprints or schemes of the crime such as a heist [3]. The essence of our attack is the sophisticated coordinated malicious payload delivery via attack irrelevant actors (e.g., web servers).

[2] **F**ail-**F**ree **D**ynamic **S**tate-**M**achine

have difficulty analyzing the sophisticated F^{2}DSM that has unbounded input and output spaces, making it difficult to reverse-engineer. Moreover, as the inputs collected from benign websites for our malware do not contain any malicious code, deep packet inspection techniques are not effective.

To demonstrate its practical applicability, we design and implement an automated malware generator, MINDGEN, which can create MASTERMIND. It takes malicious code that an attacker wants to deliver and a few attack-triggering contents from benign websites. It then creates a malware instance that will deliver the malicious code when the selected attack-triggering contents are displayed all together.

We evaluate the effectiveness of MASTERMIND by comparing it with state-of-the-art malware obfuscation techniques and malware analysis techniques including static analysis, symbolic execution, and fuzz testing tools. Our evaluation results show that MASTERMIND successfully evades state-of-the-art analysis and malware detection techniques. Our contributions are summarized as follows:

- We propose a new attack, MASTERMIND, that collects non-malicious inputs from benign websites to deliver malicious payloads without being detected. The attack is difficult to investigate forensically as it leverages benign websites and contents to mislead the investigation.
- We design and develop a reverse-engineering resilient F^{2}DSM that translates benign inputs into malicious payloads. It is capable of accepting any inputs and translating them into some outputs without causing any errors even for the inputs that are not expected by MASTERMIND (Section 3.2).
- We implement a fully automated malware creator, MINDGEN, that generates a unique MASTERMIND instance with randomized code and F^{2}DSM (Section 3.1.2).
- We create a web malware benchmark set consisting of a diverse malware samples to evaluate MASTERMIND (Section 4). We plan to release the benchmark to facilitate future research.
- We perform thorough evaluation of MASTERMIND with state-of-the-art static analysis, symbolic execution, and fuzz testing tools to show that MASTERMIND is reverse-engineering resilient (Section 4.3). Also, we compare MASTERMIND with state-of-the-art obfuscators (Section 4.1). Last but not least, we present a manual reverse-engineering attempt on MASTERMIND (Sectiom 4.4.2). The evaluation results show that analyzing MASTERMIND is fundamental challenging and difficult.

2 Motivating Example

In this section, we present a forensic investigation scenario where a forensic analyst tries to analyze and understand MASTERMIND to identify (1) malicious payload hidden in MASTERMIND and (2) malicious actors behind the attack (e.g., real identify of attackers), to demonstrate various challenges in analyzing MASTERMIND.

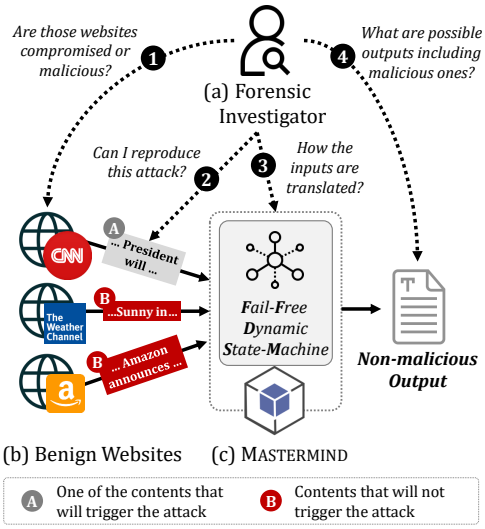


Figure 1: Forensic analysis attempt to the proposed malware. Note that as the analyst does not know correct inputs that can trigger the attack, it does not generate malicious output.

Scenario. A victim is running a website on a rented server provided by a web hosting company. One day, the victim was contacted by the company that they found a suspicious program executed malicious code on his website. The company, on behalf of the victim, hired a forensic analyst to trace back the attack and understand the details, including who are the attackers and what would be other malicious activities the suspicious program can do. The company maintains network traces for a month. As most network communications are encrypted, packet contents are not available for the analysis.

Forensic Analysis Attempt. We present a forensic analysis attempt to the motivating example. In particular, the analyst analyzes the incident from 3 different aspects: network traces, dynamic analysis of malware, and static analysis of malware.

1) Analyzing Network Traces: The forensic investigator first obtains available network logs. As the network logs do not include payloads, the investigator can only get a set of domain names and IP addresses. However, those are all benign domain and IP addresses. While a naive analyst may attribute the attack to the benign websites (i.e., answering 'Yes' to the question ❶), this is misleading because those websites are not involved in the attack campaign and the attacker intentionally used them to mislead the investigation.

2) Dynamically Analyzing the Suspicious Malware Program: Another way of understanding the attack is by running the malware program to reproduce the attack and analyze its execution and behaviors (❷). Unfortunately, the attack can be only reproduced when the collected contents from benign websites satisfy a set of predefined conditions. As the collected contents during the reproduction are different from the

predefined contents, the reproduction fails (e.g., due to the Bs in Fig. 1). The investigator may try all possible contents from those websites. However, knowing all possible content variations is difficult as websites may update at any time.

3) Statically Analyzing the Suspicious Malware Program: As dynamically executing the malware program did not reveal meaningful information, the analyst starts to analyze the malware program itself to understand how the input contents are translated (❸) as well as what it can do (e.g., what are malicious behaviors it can do) (❹).

To understand how inputs are translated into malicious payloads (❸), the analyst should reverse-engineer our complex F^{2}DSM, which is carefully designed to make it difficult to analyze. Specifically, when it translates inputs to outputs, it *always makes a transition* from any inputs and any states without any failures. This essentially makes the input space extremely large (as it can take any inputs and create outputs). Also, its output is dynamic (i.e., dependent on inputs), meaning that the output space is also large (Details in Section 3.2). In our experience, even after identifying millions of possible inputs via symbolic execution tools, the set of inputs that can trigger inputs are not found.

The analyst tries another approach: identifying possible outputs by analyzing the state-machine's outputs to recognize whether there are malicious payloads. As the output of the state-machine is *dynamically changing depending on inputs, and the state-machine does not directly include any malicious payload snippets in its plain-text form*, the attempt is not successful (Details in Section 3.2).

3 Design

In this section, we present details of our proposed malware and various design choices. We first describe an automated tool that generates MASTERMIND (Section 3.1). Then, we explain internals of MASTERMIND.

3.1 Automated MASTERMIND Generator

As shown in Fig. 5, our malware generator consists of two phases. First, it profiles various benign websites for a certain period of time to determine what content from which websites will be used to deliver malicious payloads (Section 3.1.1). Second, it creates a malware instance, specifically our novel fail-free state-machine, that translates the chosen benign contents to a chosen malicious payload (Section 3.1.2).

3.1.1 Website Profiler

Motivation. MASTERMIND operates on the inputs provided by benign websites that are *not controllable by attackers*. This design choice is crucial to hinder forensic investigation attempts. However, this naturally makes the attack more prone to fail because contents from benign websites can be changed in the future. To mitigate this and make the attack more reliable while keeping it challenging to analyze forensically, we provide a systematic way to profile benign websites to reveal good content candidates for composing MASTERMIND.

Profiling. The website profiler takes a set of benign website's webpages as input and generates a set of *ContentVector*s where each vector consists of *Content*, *Selector*, and *Statistics*. It crawls the webpage regularly (e.g., every hour) for a specified period (e.g., one month). The crawling period is configurable. For each crawled webpage, we obtain a DOM tree that only includes DOM elements that persistently appear in every crawled webpage during the profiling.

– *Content and Selector:* For each DOM element in a DOM tree, we extract common words, which we call *Content*, that appear highly frequently. Specifically, MINDGEN automatically identifies frequently appeared words via the TextRank algorithm [66], then also generates regular expressions that can extract the words automatically[1].

– *Statistics:* For each of those extracted words, we calculate (1) coverage, (2) regularity, and (3) distribution of the contents during the profiling period.

1) Coverage: Coverage represents the percentage of the DOM element appearing during the profiling. For example, if we crawl every hour for 5 days, then we have 120 (=$24*5$) data points. If a DOM element appears 100 out of 120 data points, its coverage is 83.3% (=$\frac{100}{120}$).

2) Regularity: For contents that do not appear always, the regularity represents how regularly the content will appear. To compute the regularity, for each content, we measure variances of distances between the two adjacent appearances. Specifically, given N data points, the distances between the adjacent two points, n and $n+1$, are calculated, resulting $N-1$ distances: $d_1, d_2, ..., d_{N-1}$. Then, we count the number of unique distances, denoted as $CNT_{unique\_distances}$. If all the distances are equal meaning that they are regularly appear, the value will be 1. In such case, the regularity is 1.0 (i.e., 100%). If not, we compute the regularity as follows: $1.0 - \frac{CNT_{unique\_distances}}{N-1}$.

3) Distribution: Distribution represents how the content appeared evenly during the profiling period. This is complimentary to the regularity because some content with high regularity may not evenly distributed. For instance, if some content appears at the beginning and the end of the profiling period, it may have high regularity. However, it is not evenly well distributed. Higher value in the distribution means the content has been observed evenly during the profiling period. We compute this as follows. First, given N data points as a sequence of data points D_n, we divide them into G groups (G = 24 in this paper). Essentially, the x^{th} group will include $D_x \sim D_{x+23}$. Then, we count, whether the content appear in each group and divide it by the value of G. Then, we multiply

[1]The regular expressions are detecting "patterns" (e.g., `[a-z0-9]`) to extract the words and do not include the exact words (e.g., `keyword`) in the expressions. We call it *Selector*. Hence, the regular expression itself does not reveal what exact words MASTERMIND is extracting.

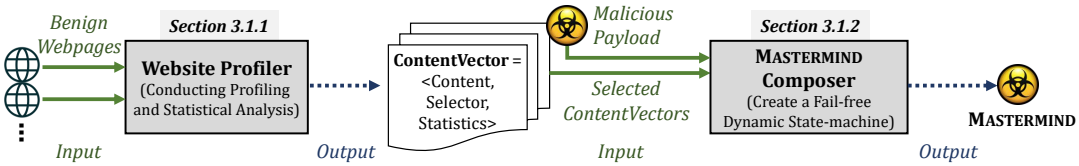


Figure 2: Malware Generator

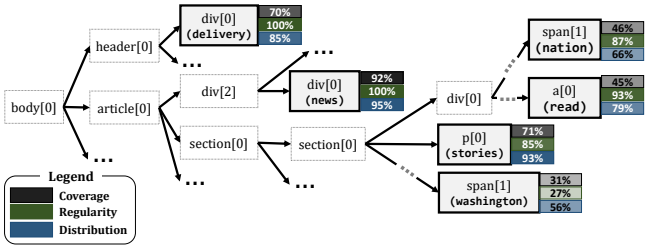


Figure 3: Selecting Contents of Interests from a DOM Tree.

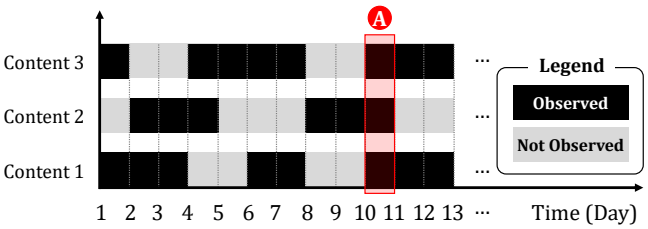


Figure 4: Aligning Contents to Control the Time of the Attack.

G by 2 and repeat the above process. After we repeat this R times (R = 5 in this paper), we add the computed value for different G and divide by R. To this end, more distributed contents will have higher distribution values.

Selecting Contents of Interest. Fig. 3 shows an interface that allows users to select contents in a DOM tree of a website. Specifically, nodes with gray borders represent DOM elements. Leaf nodes with thick borders represent contents. Leaf nodes have three colored bars displaying the coverage, regularity, and distribution values, respectively.

In addition, users can specify additional constraints for content (or modify the existing regular expression for the text extraction described above). For example, one can only list and show contents that (1) starts with a particular word, (2) ends with a particular word, and (3) matches to a user-defined regular expression, in the DOM tree.

Adjusting the Time of the Attack. Users may want to create MASTERMIND that launch the attack (i.e., deliver malicious payloads) on a specific timing. This can be achieved by selecting multiple contents with high coverage and regularity scores that do not happen at the same time.

Fig. 4 shows an example. There are three selected contents with medium to high coverage (i.e., higher than 50%) and high regularity (i.e., higher than 75%). However, as shown in the figure, the distribution of each content is disjointed mostly. There is one point, A, when all the contents appeared at the same time. By using the three selected contents, one can create MASTERMIND that only delivers at a particular timing, A. Note that this is different from a typical time-bomb malware where the time of attack is explicitly written in the malware. It is possible that MASTERMIND can achieve the same effect of the typical time-bomb by using contents that represent the current time (e.g., from a website like `https://time.gov`) and creating a translation rule that transform a particular time into malicious payload. However, this paper explores a more sophisticated time-bomb by leveraging timings of multiple contents from diverse websites. By doing so, it makes the attack more difficult to (forensically) analyze as it gives a false impression that timings of the contents may have a particular meaning. Further, users can carefully pick and align contents in order to encode bogus relationships to confuse investigation (e.g., aligning news about particular political leaders and public campaigns/operations as if they are all related to the malware instance).

Choosing Contents For Diverse Types of Attacks. As MASTERMIND collects contents from benign websites that the attack does not have control over, choosing contents is particularly important to deliver malicious payloads successfully in a stable manner while thwarting forensic analysis attempts. In the following paragraphs, we show three examples to show how to choose contents properly for different scenarios.

1) Timing-dependent (but predictable) Attack: This timing-dependent attack balances stealthiness and reliable attack delivery. Specifically, given a few predictable timing-dependent contents, users align them carefully so that the timing of the attack can be predictable and reasonably reliable. Note that selecting content with higher regularity is important to create this type of attack. For instance, one can leverage repeating content (e.g., a content of a weekly repeating event in a schedule calendar) or content related to a highly expected event (e.g., a content of thunderstorms during summer in a weather-related webpage). Content related to news articles mentioning a particular political event that is already scheduled to happen is also a good candidate for this type of attack.

2) Stealthy and Rare (but predictable) Attack: This is an attack that will be triggered rarely but in a predictable way. This type of attack is difficult to create but powerful because it is difficult to analyze as the attack would not be reproducible by forensic analysts. To create such an attack, users can select multiple contents that are rarely appear together. For instance, one may select a few contents that only happen together in a month. While this would be effective in hindering forensic

analysis attempts, this also risks the delivery attack. In other words, due to the stealthiness (i.e., rareness), the attack may not be launched when it deployed. To mitigate such uncertainty issues, one may replace some of the contents to (1) controllable contents or (2) predictable contents. Controllable contents mean that attackers can modify the contents as they wish. A twitter account page of an attacker would be an example. However, this may reveal the identity of the attackers. To be more stealthy forensically, one can leverage *predictable contents*, such as the current time and the current moon phase. The current temperature of a particular city on a weather service webpage is another example. While the weather is difficult to predict, one may expect it would reach a particular temperature (e.g., 10-celsius degree) at some point.

3) Always-active Attack: The type of attack delivers malicious payloads always. The attack will definitely happen, meaning that its success rate is very high. To create this type of attack, one can pick content with high coverage, high regularity, and high distribution scores. Note that This type of attack has a drawback: forensic analysts can easily run this malware to understand its behaviors. Essentially, this type of malware only takes advantage of the anti-forensic-analysis features of MASTERMIND (i.e., making it difficult to understand the actors behind the attack).

Handling Unexpected DOM Changes by Chaining Alternatives. A website may change its DOM structure. As MASTERMIND walks on a DOM tree to locate the selected contents of interest, this would break the malware's logic.

To handle such DOM changes, we propose a method that chains alternative contents of selected content. Specifically, for each selected content, we identify all other contents that appear as much as the originally selected content. In other words, those identified contents always appear together (or more) with the originally selected content. Then, among those identified contents, we prune out contents if their DOM paths and the originally selected content's DOM path shares more than 50% of the path so that the alternative contents can work even if 50% of the original contents' DOM path is changed.

At runtime, when MASTERMIND extracts contents from a webpage, if it fails to locate a DOM element of interest, it tries alternative contents DOM elements until it succeeds.

3.1.2 MASTERMIND Composer

MASTERMIND composer takes (1) chosen contents from the profiler and (2) malicious payloads (e.g., source code of existing malware) as input and creates MASTERMIND that translates the chosen contents into the given malicious payload via our fail-free dynamic state-machine.

Composing the Fail-free Dynamic State-machine (F^2DSM). The core of MASTERMIND is an automaton for the fail-free dynamic state-machine. The automaton consists of nodes and edges representing states and transitions between the states respectively. The edges are annotated by inputs and outputs. When a transition happens, output annotated its corresponding edge is generated.

To generate such an automaton, we first create nodes and edges for translating inputs to the given malicious payload. The resulting automaton can generate the malicious payload when the proper inputs (i.e., user chosen inputs) are provided. However, it cannot handle any other inputs.

We then add (1) bogus nodes and (2) extra edges (i.e., transitions) between all nodes. The bogus nodes are additional nodes to hinder analysis attempts of the automaton (i.e., the malicious payload creation). Extra edges are added to connect all nodes, including the nodes for malicious payload generation and bogus nodes. When we add edges to the bogus nodes, we carefully choose inputs for those edges so that inputs for all edges look *semantically similar* so that it would be difficult to distinguish edges for malicious payload generate from the bogus edges. Specifically, for each new added edge, its input is derived by perturbing its neighboring edge's input. The perturbation is done by choosing similar words from various dictionaries (e.g., for synonyms, antonyms, or alphabetically similar words) [31, 76].

Randomizing Automaton and Utility Code. To avoid detection tools that create signatures of malware from static portions of the malware (i.e., code snippets that are shared by multiple MASTERMIND instances), we randomize automaton as well as utility code snippets. Specifically, every time we create MASTERMIND, we generate a unique automaton that has randomized nodes and edges so that particular nodes and edges cannot be used to create a signature of MASTERMIND. In addition, we randomize utility code for the same reason.

To randomize automaton, we first randomly choose the number of nodes and edges (within a predetermined range such as ± 10) during the creation. Second, inputs and outputs of edges are also randomly selected.

There are utility code snippets that are responsible for parsing inputs and executing the automaton. We randomize those utility code snippets by perturbing control flow structures (e.g., changing loop structures among `for`, `foreach`, and `while`), variable names, function names, and other statements. In particular, we split a statement into multiple shorter but semantically identical statements. For instance, a statement `$v = $i + 10;` can be split into `$t = $i; $t += 5; $v = $t + 5;` to perturb the syntactic preserving the same semantic. An example of such randomization can be found in Appendix 7.1.

3.2 Fail-free Dynamic State-Machine

The fail-free dynamic state-machine (F^2DSM) is the core of MASTERMIND. It translates contents collected from benign webpages into malicious payloads.

Fail-free State-machine. A unique feature of F^2DSM is that it *does not produce any errors on any inputs* (i.e., Fail-free) whereas a typical state-machine will stuck on a state if there is

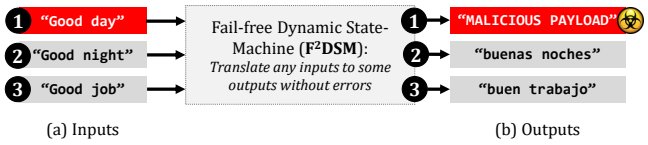


Figure 5: High-level idea of F^{2}DSM

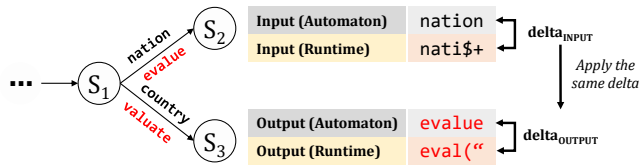
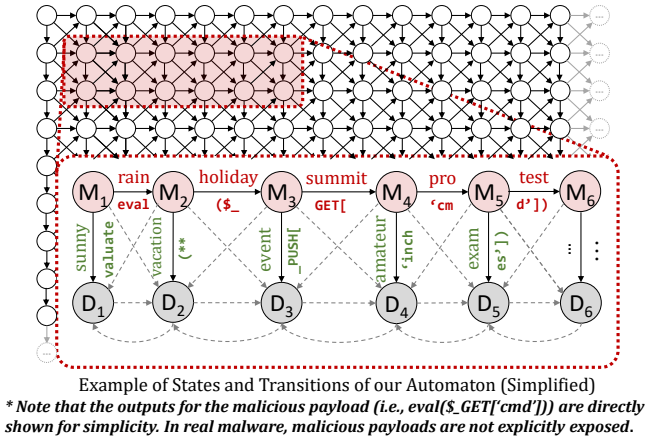


Figure 6: Dynamic Output Translation.



Example of States and Transitions of our Automaton (Simplified)

Note that the outputs for the malicious payload (i.e., eval($_GET['cmd'])) are directly shown for simplicity. In real malware, malicious payloads are not explicitly exposed.

Figure 7: F^{2}DSM example

no possible transition from the current state for an input (e.g., ill-formatted inputs). This is particularly important because if our state-machine generates errors or different behaviors (e.g., stuck on a state) on particular inputs, it essentially provides hints for attackers that the inputs are not releveant in delivering attacks in MASTERMIND. Attackers can leverage such hints to reduce the searching space for inputs that can launch attacks. Hence, F^{2}DSM is designed to force transitions from any states for any inputs. If the inputs are not relevant to malicious payload delivery, F^{2}DSM produces benign outputs (i.e., non-malicious outputs) on those inputs. Fig. 5 shows an example. The second and third inputs (❷ and ❸) lead to benign outputs while the first input (❶) leads to a malicious payload.

– *Fail-free State Transition:* F^{2}DSM generates outputs on any inputs. To achieve this, on each state, we force a transition (i.e., pick an edge) from any state on any inputs. Specifically, in F^{2}DSM, each state (i.e., node) must have multiple transitions (i.e., edges). While a naive way to implement this scheme is to add a number of edges, ideally all possible transitions, as input space is practically infinite, this is not efficient.

Instead, we force a transition (i.e., forcibly pick an edge). For example, when there is no matching edge to the given input, we calculate an edit distance [55] between the current input and each edge's input and choose the edge with the smallest edit distance (if there are multiple ties, we randomly pick one from the tied edges).

Dynamic Output Translation. The fail-free state-machine takes any inputs and makes a transition from any states. This essentially means that it extends input space to infinite (i.e., any inputs are possible in any states). However, since output space remains unchanged, it is possible to enumerate all outputs annotated on edges to recover all possible outputs.

To handle this, we design a dynamic state-machine that changes outputs according to the inputs. Specifically, assume that the state-machine takes I as input and makes a transition T^x where the transition's annotated input and output are denoted as T^x_{IN} and T^x_{OUT} respectively. This scenario essentially describes when an input that is not annotated in the state-machine is given. In this case, the fail-free state-machine makes a transition T^x because the edit distance between I and T^x_{IN} is the smallest compare to other transitions' annotated inputs (i.e., T^{others}_{IN}). We extend the output spaces, by dynamically changing T^x_{OUT} according to the current input I. Specifically, given I that is different from T^x_{IN}, and assume that the state-machine makes T^x transition, instead of generating T^x_{OUT} according to the state-machine, we generate an output computed by $delta(I, T^x_{IN}) + T^x_{OUT}$ where the *delta* function extracts differences between the two inputs in byte level and + operations represents addition operations on each byte between the two string operands. Note that this also allows us to completely remove real malicious payload from the automaton. For example, Fig. 6 shows how we can deliver part of a malicious payload `eval("` via an automaton with a transition (from S_1 to S_2) with input `nation` and output `evalue`. As shown in Fig. 6-(b), when given an input `nati$+`, the *delta* (i.e., value differences in each byte) between the current input (`nati$+`) and annotated input in the automaton (`nation`) are computed. Then, it is applied to the annotated output, `evalue`, to derive the dynamic output for the current input, `eval("`. Essentially, we use the *delta* function's output from inputs to derive a transformation function for outputs to deliver malicious payload without concretely annotating it on the state-machine. Note that the *delta* function can be redefined to leverage more advanced transformations.

– *Example:* Fig. 7 shows a simplified instance of F^{2}DSM. Due to the bogus nodes and edges, F^{2}DSM have a number of states and transitions. Fig. 7-(a) shows several states (from M_1 to M_6 and from D_1 to D_6) and transitions between the states (represented by arrows). Inputs and outputs are annotated on above and below of the arrows. Gray dotted arrows represent multiple transitions with omitted inputs and outputs (due to

the space limit in Fig. 7).

To successfully generate the predefined malicious payload `eval($_GET['cmd'])` which is a typical webshell, only six states (from M_1 to M_6) are required. Other states are added to hinder analysis attempts. For instance, the state machine will generate malicious payload when input is `rain holiday summit pro test`. Assume that the input is changed to `rain vacation ...` where `...` denotes any characters. If MASTERMIND generates an output starting with `eval(**` according to the automaton, it may leak parts of the malicious payload depending on inputs. To prevent this from happening, for an output that follows the PHP language grammar, we use a parser to check whether the generated output is ill-formatted. If so, we randomize the automaton, specifically randomizing nodes and edges, and rerun it. After the randomization, the new output becomes benign, meaning that it does not provide any hints for attackers.

Note that we choose the simple webshell to facilitate discussion. Users can choose any malware as shown in our evaluation (Section 4).

Evaluation

Implementation. We have implemented a prototype of MINDGEN in Python. It generates MASTERMIND written in PHP. The actor profiler consists of 1322 LOC and MASTERMIND including the probabilistic engine that runs the omnidirectional state machine consists of 2314 LOC, excluding lines for the automaton state descriptions. Note that while we implement the prototype in PHP, our design is general hence it can be implemented in other script languages such as JavaScript and Python.

Datasets for Evaluation. We collect 573 server-side malware from known malware collection repositories [13, 14, 16, 69, 82, 93, 95, 96, 99, 103, 105]. The samples consist of eight types as shown in Table 1. For each category, we have collected a similar number of samples (range from 61 to 79). Specifically, a webshell is malware that enables attackers to access a compromised server via a web browser that acts like a command-line interface. Backdoor is used to provide remote access (e.g., arbitrary code execution) to an infected machine for attackers. Bypassers are used to avoid detections of local or remote security mechanisms (e.g., security modules of applications and firewalls). Uploaders are used to remotely inject additional malware into victim machines. Spammers compose and send spoof/spam emails. SQLShells allow remote attackers to access databases of compromised servers, similar to webshells. A reverse shell is a type of shell that communicates back to the attackers machine from a victim machine. Flooders are used to launch Denial of Service (DoS) attacks by sending an excessive number of network packets.

Category	# of samples
Webshell	79
Backdoor	79
Bypasser	76
Uploader	76
Spammer	68
SQLShell	67
Reverse Shell	67
Flooder	61
Total	573

Table 1: Malware categorization

4.1 Comparison with Existing Obfuscators

To demonstrate that MASTERMIND is effective in delivering malware without being detected and causing false positives, we compare MASTERMIND with state-of-the-art obfuscation techniques. Specifically, we conduct two experiments. First, we use MINDGEN and existing obfuscators to hide malware and see whether the obfuscated malware samples and MASTERMIND can successfully avoid detection (Section 4.1.1). Second, we apply MINDGEN and existing obfuscators to benign PHP files to check whether the tools can cause false positives; e.g., obfuscated benign files are flagged as malware (Section 4.1.2).

Tool Selection. We use four state-of-the-art obfuscators (PHP Obfuscator [33], YAK Pro [49], PHP Best Obfuscator [23], and Simple online PHP obfuscator [56]) and three widely used malware detectors (PHP Malware Detector [92], Linux Malware Detector [35], and Shellray [68]) in this experiment. While there are also popular anti-virus software such as Microsoft Defender [26] and Norton Antivirus [27], most of them focus on detecting client side malware (e.g., client binary programs), instead of web server-side malware (e.g., PHP malware). Therefore, we do not include those anti-virus products [26, 27] because their detection rates for server-side malware are lower than the tools we choose, hence their results would be misleading.

4.1.1 Effectiveness in Hiding Malware

We show that MASTERMIND is effective in delivering malware without being detected. Out of the 573 malware shown in Table 1, we filter out malware that are not detected by each obfuscation tool we test. Specifically, PHP Malware Detector identifies 413 samples, Linux Malware Detector flags 185 samples as malware, and Shellray detects 524 samples. Then, we apply each obfuscator to obtain obfuscated malicious payload samples. Finally, we run malware detectors again on the results (i.e., obfuscated samples).

Results. Table 2 shows that that existing malware detection tools are completely unable to detect MASTERMIND (0%) while most of the malware samples obfuscated by the four tools can be detected by two detectors, PHP Malware Finder

Tool	PHP Mal. Finder		Linux Mal. Detect		Shellray	
	Malicious	Benign	Malicious	Benign	Malicious	Benign
PHP Obfuscator [33]	399/413 (96%)	161/573 (28%)	98/185 (52%)	0/573 (0%)	479/524 (91%)	0/573 (0%)
YAK Pro [49]	264/413 (63%)	139/573 (24%)	16/185 (8%)	0/573 (0%)	239/524 (45%)	1/573 (1%)
Best PHP Obfuscator [23]	412/413 (99%)	573/573 (100%)	25/185 (13%)	0/573 (0%)	505/524 (96%)	557/573 (99%)
Simple online PHP obfuscator [56]	413/413 (100%)	573/573 (100%)	0/185 (0%)	0/573 (0%)	524/524 (100%)	573/573 (100%)
MASTERMIND	0/413 (0%)	0/573 (0%)	0/185 (0%)	0/573 (0%)	0/524 (0%)	0/573 (0%)

Table 2: Evaluation results on malicious and benign samples. Gray colored cells are containing desirable values (Detected less than 10%).

Size of Payload	Average Size	# of samples	Size of MASTERMIND (Avg.)	# of nodes (Avg.)	# of edges (Avg.)
0~10 KB	2.7 KB	345	182.4 KB	603.6	4843.4
10~20 KB	14.5 KB	99	457.2 KB	2476.1	19194.7
20~30 KB	24.0 KB	39	685.0 KB	4012.8	32334.9
30~40 KB	34.7 KB	56	930.2 KB	5671.7	45659.0
40~50 KB	43.7 KB	34	1158.6 KB	7209.5	58124.1

Table 3: MASTERMIND instances breakdown by size.

(averagely 89.5%) and Shellray (83%). However, the detection rate of Linux Malware Detector (maldet) is relatively lower than other two tools. This is because maldet is not specifically designed for PHP server-side malware; it only detects 185 out of 573 malware samples (32%) without any obfuscations.

4.1.2 Experiment for Understanding False Positives

To understand whether applying MINDGEN and existing obfuscators on benign code samples would cause false positives, we collect 573 benign code snippets (i.e., files) from benign PHP programs' code bases including WordPress [34], Joomla [73], phpMyAdmin [77] and CakePHP [59]. We decide to have 573 benign files to match the same number of samples compared to the number of malware samples. Initially, none of the original benign code snippets are detected by the malware detectors. We apply MINDGEN and existing obfuscators, and run the malware detectors against the results (i.e., obfuscated samples and MASTERMIND).

Results. Table 2's Benign columns summarize the comparison results between existing obfuscators and MINDGEN on the 573 benign code snippets. We observe that *none of* MASTERMIND *is flagged as malware* (i.e., no false positive) while many benign code snippets were flagged as malware after we apply existing obfuscators. This is because they consider the signatures of those obfuscators as malicious, which can lead to the high false positive rates. Note that creating signature of MINDGEN is more challenging than doing it for existing obfuscators because MASTERMIND can be implanted to an existing program (Section 4.4.3).

4.2 Complexity of MASTERMIND

We present the complexity of MASTERMIND instances. Table 3 shows the distribution of malware samples based on the malware sizes (with an interval value of 10 KB). Specifically, the first and second column shows the size range of malware samples. The third column shows the number of samples that belong to each group. The fourth column shows the average size of MASTERMIND that deliver the malware in each group. The last two columns show the average number of nodes and edges of MASTERMIND instances.

Table 4 shows detailed statistics of 24 selected MASTERMIND instances. They are selected to represent diverse malware categories from malware samples in recent years. The first column means the ID of each malware. The second and third columns present the malware type and the date of malware sample found in the wild. The fourth and fifth columns show the size of the original malware sample and the size of the corresponding MASTERMIND. The next six columns (from the sixth to the eleventh) show detailed statistics of MASTERMIND such as the number of nodes/edges for different purposes. The next two columns present the execution time of MASTERMIND on attack-triggering inputs and non-attack-triggering inputs (i.e., side-channel attack). The last column shows the execution time of MINDGEN (for creating MASTERMIND).

We have a few observations. First, the size of the original malware and its corresponding MASTERMIND has a linear relationship. Second, as MASTERMIND's side increases, the number of nodes/edges and runtime overhead are also increasing. Third, the runtime of MASTERMIND on attack-triggering inputs and non-attack-triggering inputs are almost identical, meaning that monitoring execution time of MASTERMIND does not help understand attack-triggering inputs. Fourth, MINDGEN only takes a few second to transform (less than 10 seconds for all cases).

We observe that the number of nodes and edges are linearly increasing as the size of the malicious payload is increasing. In addition, even the samples in the smallest group (0~10 KB) have a large number of nodes and edges: 4,843 edges and 603 nodes, making it challenging to understand the underlying logic of MASTERMIND.

4.3 Analyzing MASTERMIND

We leverage existing program analysis tools to analyze MASTERMIND to show how difficult to reverse-engineer MASTERMIND with state-of-the-art program analysis techniques. In particular, we use (1) static analysis tools, (2) symbolic execution engines, and (3) fuzz testing tools to reveal malicious payload translated by F^2DSM including attack-triggering inputs. We do not include dynamic analysis as they focus on analyzing a single execution path and we assume that a malicious input is not known for analysts. Moreover, without

ID	Category	Date	Size of Payload	Size of MASTERMIND	# of nodes	# of edges	# of nodes for payload	# of edges for payload	# of nodes for non-payload	# of edges for non-payload	Runtime for payload	Runtime for non-payload	Time to transform
m0	Spammer	2019	1.3 KB	138.9 KB	301	2416	176	207	125	2209	0.02s	0.02s	6.26s
m1	Spammer	2019	1.5 KB	143.2 KB	332	2662	221	254	111	2408	0.02s	0.02s	6.27s
m2	Bypasser	2016	1.7 KB	145.8 KB	351	2811	233	276	118	2535	0.02s	0.02s	6.27s
m3	Backdoor	2017	1.9 KB	153.6 KB	406	3252	270	312	136	2940	0.02s	0.02s	6.27s
m4	Webshell	2018	2.1 KB	159.7 KB	448	3586	298	337	150	3249	0.02s	0.02s	6.28s
m5	Backdoor	2019	2.3 KB	167.1 KB	501	4009	333	361	168	3648	0.02s	0.02s	6.28s
m6	Uploader	2018	3.0 KB	184.4 KB	620	4963	413	461	207	4502	0.02s	0.02s	6.29s
m7	Backdoor	2018	4.4 KB	217.5 KB	851	6836	567	631	284	6205	0.03s	0.03s	6.32s
m8	Bypasser	2019	5.1 KB	231.3 KB	947	7596	631	722	316	6874	0.03s	0.03s	6.33s
m9	Flooder	2016	5.6 KB	237.4 KB	990	7945	659	791	331	7154	0.03s	0.03s	6.34s
m10	Flooder	2016	6.3 KB	256.5 KB	1121	8983	747	912	374	8071	0.03s	0.03s	6.36s
m11	Bypasser	2019	6.7 KB	265.8 KB	1186	9496	790	920	396	8576	0.03s	0.03s	6.36s
m12	Reverse Shell	2018	7.0 KB	286.9 KB	1327	10653	884	1028	443	9625	0.04s	0.04s	6.38s
m13	Flooder	2017	8.7 KB	333.3 KB	1643	13152	1095	1229	548	11923	0.05s	0.05s	6.42s
m14	Uploader	2016	10.6 KB	392.8 KB	2045	16395	1363	1508	682	14887	0.06s	0.06s	6.49s
m15	Webshell	2018	10.8 KB	401.6 KB	2104	16864	1402	1578	702	15286	0.06s	0.06s	6.50s
m16	Reverse Shell	2016	12.6 KB	430.1 KB	2294	18429	1529	1784	765	16645	0.07s	0.07s	6.53s
m17	SQLShell	2017	12.9 KB	435.5 KB	2335	18717	1556	1786	779	16931	0.07s	0.07s	6.54s
m18	SQLShell	2019	14.2 KB	463.9 KB	2528	20243	1685	1947	843	18296	0.08s	0.08s	6.57s
m19	Uploader	2018	17.1 KB	557.5 KB	3163	25327	2108	2444	1055	22883	0.12s	0.12s	6.68s
m20	Reverse Shell	2017	17.8 KB	578.6 KB	3307	26487	2204	2465	1103	24022	0.13s	0.13s	6.71s
m21	Spammer	2016	18.6 KB	587.9 KB	3365	26983	2243	2613	1122	24370	0.13s	0.13s	6.72s
m22	SQLShell	2017	29.1 KB	850.2 KB	5141	41259	3427	4036	1714	37223	0.27s	0.28s	7.13s
m23	Webshell	2017	47.5 KB	1334.4 KB	8419	67595	5612	6628	2807	60967	0.67s	0.68s	8.03s

Table 4: Statistics of generated MASTERMIND instances.

Program	Result	Details	Active[1]
RIPS 0.55 [28]	✗	Does not propagate taint through array index	2017
Pixy [44]	✗		2013
Eir [38]	✗		2017
Taint'em All [107]	✗		2019
TaintPHP [71]	✗	Does not support inter-procedural analysis	2016
phpSAFE [70]	✗	Does not support array	2015
Progpilot [86]	✗		2019
WAP [61]	✗		2016
phpvulhunter [72]	✗		2015
PHP Aspis [74]	✗		2011
DevBug [83]	✗		2012

✗: Failed to reveal sufficient information of MASTERMIND.
1: The last update date (e.g., last commit) of the tool.

Table 5: Analysis results via taint analysis tools.

knowing the input that can trigger attacks, dynamic analysis is ineffective in understanding hidden behaviors of MASTERMIND.

4.3.1 Static Analysis Tools

We use static analysis tools to reveal inputs that can be translated into malicious payload through F^{2}DSM. Specifically, We leverage 11 static taint analysis tools as shown in Table 5 to check whether they can infer data-flow between inputs and outputs of F^{2}DSM. We also use software bug/vulnerability detection tools that support fine-grained data-flow analysis techniques (i.e., type analysis and value-set analysis) to understand the mappings between inputs and outputs of F^{2}DSM. In particular, we choose four publicly available tools as shown in Table 6: Phantm, PHPStan, Psalm, and WeVerca. Note that their primary goal is to identify software bugs and security vulnerabilities, hence they are not directly designed to reverse-engineer malware like MASTERMIND.

Program	Result	Details	Active[1]
Phantm [50]	✗	Unable to track data-flow through the state-machine	2012
PHPStan [63]	✗	Over approximation	2019
Psalm [102]	✗	Over approximation	2019
WeVerca [39]	✗	Over approximation	2015

✗: Failed to reveal sufficient information of MASTERMIND.
1: Last commit or update of the tool.

Table 6: Analysis results via static analysis tools

Result. Table 5 and Table 6 show the result. In short, all tested static analysis tools failed to identify attack-triggering inputs of MASTERMIND. None of the tools we have tested can correctly reason F^{2}DSM's dynamic output translation (Section 3.2) and fail-free state-machine's transitions. Moreover, they also fail in analyzing other parts of F^{2}DSM. Specifically, RIPS [28], Pixy [44], Eir [38], and Taint'em All [107] fail to propagate taint tags through array index, while F^{2}DSM uses array index look-up to translate. TaintPHP [71] does not support inter-procedural analysis while MASTERMIND implements core functionalities in multiple functions and is implanted into a benign application's existing functions. Remaining tools [61, 70, 72, 74, 83, 86] in Table 5 do not support array (including array index), meaning that they do not propagate taint tags through operations on array elements.

Moreover, we observe that all of the static analysis tools we tested can only infer values of a variable as a set, meaning that they cannot infer a translation rule between a particular input and output. Instead, they can only identify a set of possible inputs and outputs. Recall that F^{2}DSM's inputs include a

Program	Result	Details	Active[1]
THAPS [43]	✗	Does not support array elements	2015
KPHP [32]	✗	Crashed due to insufficient memory	2014
Symex [67]	✗	State-explosion due to the large # of states	2015
PHPScan [101]	✗	Does not support array elements	2017

✗: Failed to reveal sufficient information of MASTERMIND.
1: Last commit or update of the tool.

Table 7: Analysis results via symbolic execution tools

Fuzzer	Result	Details	Active[1]
AFL [108]	✗	108 hours with no crash found	2019
honggfuzz [37]	✗	108 hours with no crash found	2019
libfuzzer [42]	✗	89 hours with no crash found	2019
radamsa [40]	✗	102 hours with no crash found	2019

✗: Failed to find attack triggering inputs of MASTERMIND.
1: Last commit or update of the tool.

Table 8: Analysis results via fuzz testing tools

large number of benign inputs and a few attack-triggering inputs. Hence, knowing a set of inputs do not particularly help understand MASTERMIND. Moreover, F^{2}DSM uses the edit distance algorithm to accept any inputs, meaning that its input (and output due to the dynamic output translation) space is practically infinite.

4.3.2 Symbolic Execution Tools

We use symbolic execution tools to reverse-engineer the malicious payload translation logic of MASTERMIND. THAPS [43], PHPScan [101], KPHP [32], and Symex [67] are vulnerability/software bug finding tools. To check whether it can analyze MASTERMIND or not, we implant a vulnerability/bug that is only triggered after a successful payload translation.

Result. As shown in Table 7, KPHP [32] crashed after running 7 hours 17 minutes, due to the insufficient memory. This is, in part, because our automaton includes a number of nodes and edges, leading to the state-explosion problem [18,22,100]. THAPS [43] and PHPScan [101] failed to analyze MASTERMIND, as they do not support array elements. MASTERMIND heavily utilizes array elements for its translation. Symex failed to analyze MASTERMIND because of the large number of nodes and edges causing state-explosion.

4.3.3 Fuzz Testing Tools

Fuzz testing tools aim to discover software bugs or vulnerabilities by inputting random data and observing observable software failures (e.g., crashes). We use existing fuzz testing tools to find attack-triggering inputs. As fuzz testing tools look for software crashes, we create MASTERMIND instances to deliver malicious payloads that crash the malware themselves so that the fuzz testing tools can detect when the malicious payloads are successfully delivered.

We choose four publicly available fuzzers: AFL [108], honggfuzz [37], libfuzzer [42], and radamsa [40]. Note that there are other fuzz testing tools we do not use. First, Phuzzy [80] and Minerva [51] are focusing fuzzing function invocations in PHP programs while we want to fuzz the inputs of MASTERMIND. LangFuzz [41] and Kameleon Fuzz [30] are not selected as they are not publicly available.

Note that the fuzzers we have chosen are not particularly designed for PHP programs. Hence, we apply them to MASTERMIND directly with some guided inputs of MASTERMIND so that the fuzzers can test MASTERMIND with randomized inputs. Specifically, we run the fuzzers with a seed input that we extracted from MASTERMIND's source code (i.e., from the array that contains input keywords). Also, `AFL`, `honggfuzz`, and `libfuzzer` support *dictionaries* [57] which contains a set of input keywords that can be used when a fuzzer perturbs inputs. We use ten randomly picked inputs extracted from the input keyword array in the MASTERMIND's instance. `radamsa` does not support dictionaries, hence we run them with the seed inputs.

While the choosen fuzzers are not specialized to PHP programs, hence they might be inefficient, we find that those are most powerful tools even for PHP programs. Note that we have verified that those fuzzers can find crashes in PHP programs when we provide a simple malware program that delivers the same crashing payload we used in our experiment.

All four fuzzers were not able to identify any crashes (i.e., any attack-triggering inputs). First, we run `AFL` for 4 days 12 hours. The code coverage metric reported by `AFL` reaches a fixed point, 13.66%, for more than 31 hours. Second, `honggfuzz` runs more than 4 days without any success. It reached 3.06% code coverage after 55 hours (2 days 7 hours). `libfuzzer` reports the total number of code blocks or edges covered via the *cov* option [57], which reached a fixed point, 3593, after 76 hours (3 days 4 hours). `radamsa` does not provide any metrics to evaluate code coverage. We ran it for 102 hours (4 days 6 hours) without any crashes.

4.4 Case Study

In addition to the motivation example presented in Section 2, we show an end-to-end scenario of attack creation via MINDGEN (Section 4.4.1). We also present a manual reverse-engineering attempt to understand MASTERMIND by manually guessing inputs (Section 4.4.2).

4.4.1 Attack Example

We show a scenario of creating a MASTERMIND instance. Then, we measure the success of the attack.

MASTERMIND Construction. We choose five benign websites for delivering attack-triggering contents: Fox News [25], Earthquake Track [6], Kimbell Museum [2], Weather in New York [97], and Chrome Release Blog [36]. For each website,

we conduct profiling for 20 days (from May 7th 2019 to May 27th 2019). Based on the profiling result, we choose a DOM element that has 100% stability. The third column of Table 9 shows DOM paths of the chosen DOM elements. Due to the space, we omit some DOM elements' names within the DOM paths. The fourth column shows the attack-triggering contents. Specifically, the attack will be launched when the chosen DOM's content is aligned with the attack-triggering contents. Note that determining whether the attack-triggering contents are satisfied or not is encoded as a part of the omnidirectional state machine. MASTERMIND does not include predicates that explicitly check the attack-triggering contents shown in Table 9 (e.g., `if (strstr($contents, "charged")) {...}`).

Attack Expectation and Evaluation. Given the constructed MASTERMIND, we evaluate whether MASTERMIND works as expected during another time frame, from May 28th 2019 to June 17th 2019, which we call evaluation period. Table 10 shows measured frequencies of chosen DOM elements containing attack-triggering contents. The second and third columns show measured frequencies during profiling and evaluation respectively. The fourth column shows the differences between profiling and evaluation.

Observe that the measured frequency differences (the fourth column) are not significant (less than ±5%). The last row (combined) shows the frequency of all the attack-triggering contents satisfied together. They both are 0.1%, and it is observed once during the profiling and evaluation, respectively. Note that all the contents we picked have 100% stability during both profiling and evaluation.

The results show that while there is uncertainty as the chosen benign websites' contents can change arbitrary, the variance is not significant in practice. As a result, the attack successfully delivered during the evaluation period as expected.

Analyzing the Sample. As discussed in Section 4.2, existing analysis techniques are ineffective in revealing attack-triggering inputs and malicious payloads encoded in MASTERMIND. In addition, the sample is collecting attack-triggering contents from benign websites. An attempt to understand the motive behind MASTERMIND would be difficult. For instance, the contents used by MASTERMIND for translation do not have any relationship between the delivered payloads. More importantly, the attack-triggering contents are deliberately chosen to confuse the motive. In other words, while the attack will happen when *Weather in New York* website [97] displays `thunderstorms`, the fact that there are thunderstorms in New York and the website are not related to the attack.

4.4.2 Input/Output Space Exploration

In this case study, we present an attempt to analyze MASTERMIND by manually exploring possible inputs and outputs. Specifically, an analyst may go through this exploration process to identify possible malicious outputs. We start from two seed inputs: (1) a non-attack-triggering input obtained by running the MASTERMIND. This seed input does not lead to an attack. (2) an attack-triggering input. We show that even a small change from an attack-triggering input would not provide useful hints for analysis.

	Website	Chosen DOM element	Triggering Contents
C1	**Fox News** [25]	#wrapper > ... > article:nth-child(3) > ... > h2	Contains "`charged`"
C2	**Earthquake Track** [6]	#content > ... > ul > li:nth-child(1) > div	Matches with "`this week: * in namie, (*), japan`"
C3	**Kimbell Museum** [2]	body > ... > div.event-teaser-text > h3	Starts with "`happy hour`"
C4	**Weather in New York** [97]	#twc-scrollabe > ... > tr:nth-child(15) > td.description	Ends with "`thunderstorms`"
C5	**Chrome Release Blog** [36]	#Blog1 > ... > span.labels > a:nth-child(1)	Starts with "`beta updates`"

Table 9: Attack triggering conditions for the example attack.

	Profiling	Evaluation	Difference
C1	4.7%	6.7%	+2%
C2	17.2%	17.8%	+0.6%
C3	25.3%	28.9%	+3.6%
C4	39.2%	43.5%	+4.3%
C5	44.8%	42.7%	-2.1%
Combined	0.1%	0.1%	0%

Table 10: Measured frequency metrics during profiling and evaluation period of the example attack.

Exploration from a Non-attack-triggering Input. Table 11 shows the input and output pairs from the experiment. The first row (shaded row) shows the seed input. From the second row, we modify inputs by guessing a possible word to replace. We assume that a sensible attacker may choose each word for replacement within a set of similar words found in the malware (e.g., words annotated in our automaton's edges).

Observe that even changing a single word to a similar word will significantly change the output. More importantly, changing the same word into a different word results in a completely different output. This is because, changing a single word will completely change how the input traverses the automaton, creating a cascading effect on the output.

Exploration from an Attack-Triggering Input. Table 12 shows the exploration results from an attack-triggering input. The first row shows the seed attack-triggering input and the subsequent rows show the perturbed inputs and their corresponding outputs.

Note that changing a single word leads to a completely different output. In this experiment, we want to show that MASTERMIND does not give out hints that the input is close to the attack-triggering input, even it is indeed very close. Observe that changing other words does not provide particular hints to guess the original attack-triggering input. Moreover, MASTERMIND may include multiple attack-triggering inputs. The

Input	Output
Alabama PM Showers European Chrome Beta for Android Frustrated	Un grande guía, dirigida por turísticos voluntarios altamente capacitados.
Baltimore PM Showers European Chrome Beta for Android Frustrated	**Nueva Zelanda descubrió guías turísticos voluntarios altamente capacitados, es.**
Virginia PM Showers European Human Presence Partly Cloudy	**Buen artículo fue Mike Ashley semana con digital el;**
Alabama **AM** Showers European Chrome Beta for Android Frustrated	Un **trabajo puede necesitar Cuando dos arte y el?**
Alabama PM **Thunderstorm** European Chrome Beta for Android Frustrated	Un **trabajo es parte de las propuestas para estudiar.**

Table 11: Exploration from a Non-attack-Triggering Input.

Input	Output
Fires Thunderstorms Beta Updates Fridays at Trinity very poor Sweden	exec('iptables -t nat -A PREROUTING -p tcp --dport 80 -j DNAT --to-destination 192.168.42.10:80')
Water Thunderstorms Beta Updates Fridays at Trinity very poor Sweden	**el aire en Dorchester alcanza un alto nivel de la.**
Ice Thunderstorms Beta Updates Fridays at Trinity very poor Sweden	**La historia de tierra y gatos. turistas voluntarios altamente capacitados!**
Fires **Sunny** Beta Updates Fridays at Trinity very poor Sweden	**animal como ese lugar La policía como una comunidad en.**
Fires Thunderstorms **Stable** Updates Fridays at Trinity very poor Sweden	**avión volador debe ser establecido Dorset 5:30 pm, cómo el;**

Table 12: Exploration from an Attack-Triggering Input.

result also indicates that *knowing one of the attack-triggering inputs does not help identify other attack-triggering inputs*.

4.4.3 Implanting MASTERMIND

F^{2}DSM is highly complex and hence one may believe that it is distinctive from other applications. One may concern that it can be easily detected by modeling such distinctiveness of the F^{2}DSM (e.g., modeling code snippets for a complex automaton). In this case study, we show that it is not straightforward to precisely model and challenging to detect MASTERMIND because MASTERMIND can be implanted into a benign program. Specifically, we select an existing program `PhpSpreadsheet` [78], that converts different spreadsheet file formats. We implant F^{2}DSM into the program. The implanted program includes software components (e.g., functions and variables) for the original program as well as for MASTERMIND. As variables and functions are shared between the original and MASTERMIND parts, *it is challenging to create a signature that only includes code snippets for* MASTERMIND *but not benign code snippets*.

Fig. 8 shows code snippets from the existing program `PhpSpreadsheet` (Fig. 8-(a)) and a modified program with MASTERMIND in it (Fig. 8-(b)). The highlighted portions are the modified parts to embed MASTERMIND. In particular, inside the array, `category` is used to describe the nodes, `functionCall` is for inputs and outputs of the edges of automaton, and `argumentCount` denotes edges between nodes. Observe that while Fig. 8-(a) and Fig. 8-(b) are different, they are syntactically similar, making it challenging to distinguish. As a result, if a signature is generated from the parts that are not changed, it would cause false positives (i.e., flagging benign application as malware).

In addition to this, we add 41 statements, modify 7 statements (e.g., modify variables names in them), modify 5 functions (e.g., adding and removing parameters and returns), add 11 new variables and change 5 variables (e.g., change their types). As those added/modified statements and functions can be used to create a signature to detect MASTERMIND, when we insert those statements, we randomly spread those in existing functions in addition to the statement randomization scheme described in Section 3.1.2.

5 Discussion

Availability of Attack Triggering Inputs. We assume that an analyst does not know the attack-triggering input, and the goal of the analyst is to identify attack-triggering input from the malware. If an analyst knows the attack-triggering inputs, malicious payload hidden in MASTERMIND can be exposed by executing it directly. In practice, precisely identifying attack-triggering inputs of MASTERMIND is particularly challenging. This is because, first, MASTERMIND collects inputs from benign websites. Even if there are available network traffic logs, it is difficult to pinpoint which inputs were used by MASTERMIND because many other network queries to the benign websites might be issued while MASTERMIND is active. Moreover, MASTERMIND can include multiple nested malicious payloads (Section 3.1.1). In such case, knowing an attack triggering input does not help reveal other attack triggering inputs.

Manually Analyzing MASTERMIND. F^{2}DSM contains an array that includes all input words. It is possible that an analyst may bruteforce all possible inputs of MASTERMIND. However, as the input words are not ordered, such a bruteforce approach would not work as it needs to figure out the exact order of inputs. As shown in Section 4, it requires significant effort to identify attack-triggering inputs.

Mitigation. To detect and prevent MASTERMIND, an automated analysis techniques specialized for F^{2}DSM is needed. In other words, an analysis engine that can understand and explore a finite state-machine may reveal malicious payloads hidden in MASTERMIND. However, due to the large number of states and transitions coupled with the forced transition via edit distance and randomized automaton scheme (Section 3.2), it still requires significant effort to reverse-engineer MASTERMIND even with an analysis specialized to a state-machine. A real-time detector and monitoring tool can prevent MASTERMIND from damaging the victim system. However, it is limited to preventing damages. Investigating MASTERMIND and identifying malicious actors behind the attack is still challenging. Note that creating signature of MASTERMIND is not trivial as it can be embedded into benign applications as shown in Section 3.2.

Generality of the Attack. While we implement our proto-

```php
private static $phpSpreadsheetFunctions = [
  'PRICE' => [
    'category' => Category::CATEGORY_FINANCIAL,
    'functionCall' => [Financial::class, 'PRICE'],
    'argumentCount' => '6,7',
  ],
  'PRICEDISC' => [
    'category' => Category::CATEGORY_FINANCIAL,
    'functionCall' => [Financial::class, 'PRICEDISC'],
    'argumentCount' => '4,5',
  ],
  'PRICEMAT' => [
    'category' => Category::CATEGORY_FINANCIAL,
    'functionCall' => [Financial::class, 'PRICEMAT'],
    'argumentCount' => '5,6',
  ],
  ...
];
```

(a) Original Program

```php
private static $phpSpreadsheetFunctions = [
  'PRICE' => [
    'category' => 0,
    'functionCall' => [Financial::class, 'rain|holiday|physics'],
    'argumentCount' => '6,17;0,23,79;15,56,93;',
  ],
  'QUEST' => [
    'category' => 1,
    'functionCall' => [Financial::class, 'summary|summit|summer|sunny'],
    'argumentCount' => '8,13;18;63,77;2,5;',
  ],
  'stem' => [
    'category' => 2,
    'functionCall' => [Financial::class, 'exam|holy|right'],
    'argumentCount' => '56,113,92;53,47,29;13,70;',
  ],
  ...
];
```

(b) Modified Program with MASTERMIND embedded

Figure 8: Embedding MASTERMIND into an existing software.

type in PHP, the idea is general and can be implemented in other script languages that support dynamic constructs such as `eval()`. For instance, JavaScript and Python are popular script languages which support such dynamic construct primitives. We leave exploring possibilities of leveraging other script languages to future work.

Size of MASTERMIND. MASTERMIND increases the size of original payloads as we add F^{2}DSM to translate benign contents into malicious payloads. However, the size are still reasonable considering the speed of modern network infrastructure. Moreover, this can be mitigated by applying PHP compression techniques [60]. We tried to compress MASTERMIND instances, and it generally reduces the size of MASTERMIND significantly (the compressed malware only takes less than 20% of the original MASTERMIND's size).

6 Related Work

Obfuscation Techniques. There exists a line of work in obfuscation to hide malicious code leveraging opaque predicates [24, 65, 88], code insertion/replacement [54, 79], encryptions [87, 104], hardware primitives [20, 85]. However, opaque predicates can be detected and removed via advanced program analysis techniques [62]. Dummy code snippets inserted into an existing program can be identified and removed via dependency analysis such as taint analysis [28, 38, 44, 61, 70–72, 74, 83, 86, 107]. Obfuscation techniques can also be identified by translating a program into an abstracted form (e.g., AST or Intermediate Representation). Data obfuscations (e.g., encrypting code sections and decryption them at runtime) are easily handled by dynamic analysis [17, 58, 91]. Approaches require particular hardware support are difficult to be used in real-world malware, as many systems may not satisfy the hardware requirement. Unlike them, MASTERMIND is challenging to be analyzed by static, symbolic, and dynamic analysis including fuzz testing tools as shown in Section 4. It does not require any particular hardware or software.

Advanced Malware Analysis Techniques. A group of research [10–12, 21, 32, 45, 46, 48, 52, 58, 64, 81, 84, 90, 94] tries to detect and analyze malware. In particular, dynamic analysis based forced execution techniques [47] aim to handle evasive JavaScript malware. They forcibly drive execution into every branch even if the branch condition is not satisfied. While they are effective in detecting malware that hides malicious code behind sophisticated predicates, it is not effective in expose malicious payload in MASTERMIND because it is encoded as states and transitions of the omnidirectional state machine. There are also forensic analysis and network traffic analysis approaches that analyze the causal relationship between network and system events [11, 94]. For such techniques, MASTERMIND is difficult to analyze as it gets all inputs from common benign websites where many other applications and systems may access them when MASTERMIND is active. As a result, understanding who are the actors behind the attack is particularly challenging. There are approaches that detect common patterns of malware [46, 48]. While they are effective in traditional malware, MASTERMIND can evade such techniques as MASTERMIND can be implanted into existing programs as shown in Section 3.2. Finally, there are static, symbolic, and fuzz testing tools for malware analysis that can reveal hidden malicious behaviors of malware. As discussed and shown in Section 4, MASTERMIND is resilient to such advanced malware analysis techniques.

7 Conclusion

In this paper, we present MASTERMIND, a new type of attack that impose fundamental challenges to post-mortem forensic analysis including program analysis techniques. We design and implement a novel F^{2}DSM that can effectively thwart forensic analysis attempts via advanced program analysis techniques. With MASTERMIND, malicious payloads can be delivered with virtually no traces that can be attributed to the real malicious actors. We evaluate our prototype by creating MASTERMIND instances from 573 real-world malware samples and applyying various program analysis tools and conducting forensic analysis. The results show that MASTERMIND is effective in secretly delivering malicious payloads without being detected.

References

[1] Command and control server in social media (Twitter, Instagram, Youtube + Telegram), 2018. https://medium.com/@woj_ciech/command_and_control_server_in_social_media_twitter_instagram_youtube_telegram-5206ce763950/. (2018).

[2] Calendar | Kimbell Art Museum, 2019. https://www.kimbellart.org/calendar. (2019).

[3] Death of Brian Wells - Wikipedia, 2019. https://en.wikipedia.org/wiki/Death_of_Brian_Wells. (2019).

[4] ESPN: Serving sports fans. Anytime. Anywhere., 2019. https://www.espn.com/. (2019).

[5] Houston Rockets (@HoustonRockets) / Twitter, 2019. https://twitter.com/HoustonRockets. (2019).

[6] Today's Earthquakes in East Coast Of Honshu, Japan, 2019. https://earthquaketrack.com/r/east-coast-of-honshu-japan/recent. (2019).

[7] Trinity Church Boston, 2019. https://www.trinitychurchboston.org/. (2019).

[8] USA TODAY: Latest World and US News, 2019. https://www.usatoday.com/. (2019).

[9] Moataz AbdelKhalek and Ahmed Shosha. Jsdes: An automated de-obfuscation system for malicious javascript. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, pages 80:1–80:13, New York, NY, USA, 2017. ACM.

[10] Hyrum S Anderson, Anant Kharkar, Bobby Filar, David Evans, and Phil Roth. Learning to evade static pe machine learning malware models via reinforcement learning. *arXiv preprint arXiv:1801.08917*, 2018.

[11] Azeem Aqil, Ahmed OF Atya, Trent Jaeger, Srikanth V Krishnamurthy, Karl Levitt, Patrick D McDaniel, Jeff Rowe, and Ananthram Swami. Detection of stealthy tcp-based dos attacks. In *MILCOM 2015-2015 IEEE Military Communications Conference*, pages 348–353. IEEE, 2015.

[12] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401. IEEE, 2008.

[13] bartblaze. GitHub - bartblaze/PHP-backdoors: A collection of PHP backdoors, 2019. https://github.com/bartblaze/PHP-backdoors.

[14] BDLeet. GitHub - BDLeet/public-shell: Some Public Shell, 2016. https://github.com/BDLeet/public-shell.

[15] Michela Becchi and Patrick Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT conference*, page 1. ACM, 2007.

[16] BlackArch. GitHub - BlackArch/webshells: Various webshells, 2019. https://github.com/BlackArch/webshells.

[17] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer, 2008.

[18] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2):142–170, 1992.

[19] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[20] Haibo Chen, Liwei Yuan, Xi Wu, Binyu Zang, Bo Huang, and Pen-chung Yew. Control flow obfuscation with information flow tracking. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 391–400. ACM, 2009.

[21] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant. Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 32–46. IEEE, 2005.

[22] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1–30. Springer, 2011.

[23] Pipsomania Co. Best PHP Obfuscator, 2018. http://www.pipsomania.com/best_php_obfuscator.do.

[24] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196. ACM, 1998.

[25] Fox Corporation. Fox News, 2019. https://www.foxnews.com/. (2019).

[26] Microsoft Corporation. Microsoft Defender Advanced Threat Protection, 2019. https://docs.microsoft.com/en-us/windows/security/threat-protection/microsoft-defender-atp/microsoft-defender-advanced-threat-protection.

[27] Symantec Corporation. Official Site | Norton™ - Antivirus & Anti-Malware Software, 2019. https://us.norton.com/.

[28] Johannes Dahse and Jörg Schwenk. Rips-a static source code analyser for vulnerabilities in php scripts. *Retrieved: February*, 28:2012, 2010.

[29] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. Deep packet inspection using parallel bloom filters. In *11th Symposium on High Performance Interconnects, 2003. Proceedings.*, pages 44–51. IEEE, 2003.

[30] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. Kameleonfuzz: evolutionary fuzzing for black-box xss detection. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 37–48. ACM, 2014.

[31] dwyl. A text file containing 479k English words, 2019. https://github.com/dwyl/english-words.

[32] Daniele Filaretti and Sergio Maffeis. An executable formal semantics of php. In *European Conference on Object-Oriented Programming*, pages 567–592. Springer, 2014.

[33] Maurice Fonk. GitHub - naneau/php-obfuscator: an "obfuscator" for PSR/OOp PHP code, 2019. https://github.com/naneau/php-obfuscator.

[34] WordPress Foundation. WordPress, 2019. https://wordpress.com/.

[35] R fx Networks. Linux Malware Detect, 2019. https://www.rfxn.com/projects/linux-malware-detect/.

[36] Google. Chrome Releases, 2019. https://chromereleases.googleblog.com/. (2019).

[37] Google. GitHub - google/honggfuzz: Security oriented fuzzer with powerful analysis options. Supports evolutionary, feedback-driven fuzzing based on code coverage (software- and hardware-based), 2019. https://github.com/google/honggfuzz.

[38] Heilan Yvette Grimes. Eir - static vulnerability detection in php applications. 2015.

[39] David Hauzar and Jan Kofroň. Weverca: Web applications verification for php. In *International Conference on Software Engineering and Formal Methods*, pages 296–301. Springer, 2014.

[40] Aki Helin. GitLab - Aki Helin/radamsa: A general-purpose fuzzer, 2019. https://gitlab.com/akihe/radamsa.

[41] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012.

[42] LLVM Compiler Infrastructure. libfuzzer: A library for coverage-guided fuzz testing, 2017.

[43] Torben Jensen, Heine Pedersen, Mads Chr Olesen, and René Rydhof Hansen. Thaps: automated vulnerability scanning of php applications. In *Nordic conference on secure IT systems*, pages 31–46. Springer, 2012.

[44] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 6–pp. IEEE, 2006.

[45] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 18(5):861–907, 2010.

[46] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Revolver: An automated approach to the detection of evasive web-based malware. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 637–652, 2013.

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

[48] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Detecting malicious code by model checking. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 174–187. Springer, 2005.

[49] Pascal KISSIAN. YAK Pro: Php Obfuscator, 2019. https://www.php-obfuscator.com/.

[50] Etienne Kneuss, Philippe Suter, and Viktor Kuncak. Phantm: Php analyzer for type mismatch. In *FSE'10 Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, number CONF, 2010.

[51] Mateusz Kocielski. GitHub - LogicalTrust/minerva_lib: polish fuzzy lop - fuzzer for libraries, 2018. https://github.com/LogicalTrust/minerva_lib.

[52] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European Signal Processing Conference (EUSIPCO)*, pages 533–537. IEEE, 2018.

[53] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 339–350. ACM, 2006.

[54] Byoungyoung Lee, Yuna Kim, and Jong Kim. binob+: a framework for potent and stealthy binary obfuscation. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 271–281. ACM, 2010.

[55] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, February 1966.

[56] Robert Lie. Simple online PHP obfuscator: encodes PHP code into random letters, numbers and/or characters, 2019. https://www.mobilefish.com/services/php_obfuscator/php_obfuscator.php.

[57] LLVM. libFuzzer – a library for coverage-guided fuzz testing. — LLVM 10 documentation, 2019. https://llvm.org/docs/LibFuzzer.html.

[58] Jian Mao, Jingdong Bian, Guangdong Bai, Ruilong Wang, Yue Chen, Yinhao Xiao, and Zhenkai Liang. Detecting malicious behaviors in javascript applications. *IEEE Access*, 6:12284–12294, 2018.

[59] Larry Masters. CakePHP: The Rapid Development Framework for PHP, 2019. https://cakephp.org/.

[60] Maurits van der Schee. compress.php, 2019. https://github.com/mevdschee/compress.php.

[61] Ibéria Medeiros, Nuno F Neves, and Miguel Correia. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the 23rd international conference on World wide web*, pages 63–74. ACM, 2014.

[62] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 757–768. ACM, 2015.

[63] Ondřej Mirtes. GitHub - phpstan/phpstan: PHP Static Analysis Tool, 2019. https://github.com/phpstan/phpstan.

[64] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *2007 IEEE Symposium on Security and Privacy (SP'07)*, pages 231–245. IEEE, 2007.

[65] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430. IEEE, 2007.

[66] Paco Nathan. Pytextrank, a python implementation of textrank for text document nlp parsing and summarization. https://github.com/ceteri/pytextrank/, 2016.

[67] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Auto-locating and fix-propagating for html validation errors to php server-side code. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 13–22. IEEE Computer Society, 2011.

[68] Nimbusec. Shellray: A PHP webshell detector, 2019. https://shellray.com/.

[69] nixawk. GitHub - nixawk/fuzzdb: Web Fuzzing Discovery and Attack Pattern Database, 2018. https://github.com/nixawk/fuzzdb.

[70] Paulo Jorge Costa Nunes, José Fonseca, and Marco Vieira. phpsafe: A security analysis tool for oop web application plugins. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 299–306. IEEE, 2015.

[71] Oswaldo Olivo. GitHub - olivo/TaintPHP: Static Taint Analysis for PHP web applications, 2016. https://github.com/olivo/TaintPHP.

[72] OneSourceCat. GitHub - OneSourceCat/phpvulhunter: A tool that can scan php vulnerabilities automatically using static analysis methods, 2015. https://github.com/OneSourceCat/phpvulhunter.

[73] Inc. Open Source Matters. Joomla: Content Management System (CMS), 2019. https://www.joomla.org/.

[74] Ioannis Papagiannis, Matteo Migliavacca, and Peter Pietzuch. Php aspis: using partial taint tracking to protect against injection attacks. In *2nd USENIX Conference on Web Application Development*, volume 13, 2011.

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

[76] PHP. PHP: Pspell Functions, 2019. https://www.php.net/manual/en/ref.pspell.php.

[77] phpmyadmin. GitHub - phpmyadmin/phpmyadmin: A web interface for MySQL and MariaDB, 2019. https://www.joomla.org/.

[78] PHPOffice. GitHub - PHPOffice/PhpSpreadsheet: A pure PHP library for reading and writing spreadsheet files, 2018. https://github.com/PHPOffice/PhpSpreadsheet.

[79] Igor V Popov, Saumya K Debray, and Gregory R Andrews. Binary obfuscation using signals. In *USENIX Security Symposium*, pages 275–290, 2007.

[80] Nikita Popov. GitHub - nikic/Phuzzy: Fuzzer for PHP in PHP), 2012. https://github.com/nikic/Phuzzy.

[81] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. A semantics-based approach to malware detection. *ACM SIGPLAN Notices*, 42(1):377–388, 2007.

[82] Ridter. GitHub - Ridter/Pentest, 2019. https://github.com/Ridter/Pentest.

[83] Dewhurst Ryan. Implementing basic static code analysis into integrated development environments (ides) to reduce software vulnerablitilies. *A Report submitted in partial fulfillment of the regulations governing the award of the Degree of BSc (Honours) Ethical Hacking for Computer Security at the University of Northumbria at Newcastle*, 2012, 2011.

[84] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *2010 IEEE Symposium on Security and Privacy*, pages 513–528. IEEE, 2010.

[85] Sebastian Schrittwieser, Stefan Katzenbeisser, Peter Kieseberg, Markus Huber, Manuel Leithner, Martin Mulazzani, and Edgar Weippl. Covert computation: Hiding code in code for obfuscation purposes. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 529–534. ACM, 2013.

[86] Design Security. GitHub - designsecurity/progpilot: A static analysis tool for security, 2016. https://github.com/designsecurity/progpilot.

[87] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[88] Brendan Sheridan and Micah Sherr. On manufacturing resilient opaque constructs against static analysis. In *European Symposium on Research in Computer Security*, pages 39–58. Springer, 2016.

[89] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. *ACM SIGCOMM Computer communication review*, 45(4):213–226, 2015.

[90] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. Unearthing stealthy program attacks buried in extremely long execution paths. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 401–413. ACM, 2015.

[91] Guillermo Suarez-Tangil, Juan E Tapiador, Flavio Lombardi, and Roberto Di Pietro. Thwarting obfuscated malware via differential fault analysis. *Computer*, 47(6):24–31, 2014.

[92] NBS Systems. GitHub - nbs-system/php-malware-finder: Detect potentially malicious PHP files, 2019. https://github.com/nbs-system/php-malware-finder/.

[93] tanjiti. GitHub - tanjiti/webshellSample: Webshell sample for WebShell Log Analysis, 2018. https://github.com/tanjiti/webshellSample.

[94] Teryl Taylor, Xin Hu, Ting Wang, Jiyong Jang, Marc Ph Stoecklin, Fabian Monrose, and Reiner Sailer. Detecting malicious exploit kits using tree-based similarity searches. In *proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 255–266. ACM, 2016.

[95] tdifg. GitHub - tdifg/WebShell: WebShell Collect, 2016. https://github.com/tdifg/WebShell.

[96] tennc. GitHub - tennc/webshell: A webshell open source project, 2019. https://github.com/tennc/webshell.

[97] LLC the Weather Group. New York, NY 10-Day Weather Forecast - The Weather Channel, 2019. https://weather.com/weather/tenday/l/New+York+NY+10010:4:US. (2019).

[98] Ramtine Tofighi-Shirazi, Maria Christofi, Philippe Elbaz-Vincent, and Thanh-ha Le. Dose: Deobfuscation based on semantic equivalence. In *Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop*, SSPREW-8, pages 1:1–1:12, New York, NY, USA, 2018. ACM.

[99] John Troon. GitHub - JohnTroony/php-webshells: Common php webshells, 2016. https://github.com/JohnTroony/php-webshells.

[100] Antti Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the Volumes Are Based on the Advanced Course on Petri Nets*, pages 429–528, London, UK, UK, 1998. Springer-Verlag.

[101] Bart van Arnhem. GitHub - bartvanarnhem/phpscan: Symbolic execution inspired PHP application scanner for code-path discovery, 2017. https://github.com/bartvanarnhem/phpscan.

[102] Vimeo. GitHub - vimeo/psalm: A static analysis tool for finding errors in PHP applications, 2019. https://github.com/vimeo/psalm.

[103] VirusShare. VirusShare, 2019. https://virusshare.com/.

[104] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. Linear obfuscation to combat symbolic execution. In *European Symposium on Research in Computer Security*, pages 210–226. Springer, 2011.

[105] xl7dev. GitHub - xl7dev/WebShell: Webshell && Backdoor Collection, 2017. https://github.com/xl7dev/WebShell.

[106] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A generic approach to automatic deobfuscation of executable code. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 674–691, Washington, DC, USA, 2015. IEEE Computer Society.

[107] Quan Yang. GitHub - quanyang/Taint-em-All: A taint analysis tool for the PHP language, 2019. https://github.com/quanyang/Taint-em-All.

[108] Michal Zalewski. american fuzzy lop, 2019. http://lcamtuf.coredump.cx/afl/.

Appendix

7.1 Example of MASTERMIND variants

Fig. 9 shows an example of two variants generated. The highlighted lines (Lines 10~13 and 22~25 in Fig. 9-(a) and Lines 35~38, 44~45, and 54~55 in Fig. 9-(b)) are code snippets for MASTERMIND. Lines that are not highlighted are code snippets from `PhpSpreadsheet`.

Observe that MASTERMIND identifies and reuses an existing function, `_parseFormula`, to implant the code snippets. Also, the code snippets are embedded in different places between the variants. Moreover, some variables from `PhpSpreadsheet` (e.g., `$opCharacter`) are reused by MASTERMIND, making it more challenging to analyze.

In addition, as mentioned in Section 3.1.2, statements are randomized while preserving their semantics. Specifically, `array_search` (❶ at line 13) is replaced with `array_keys` (Ⓐ at line 38). Parameters for the function are also changed accordingly. Similarly, we replace `array_merge` (❸ at line 24) with `+` operator (Ⓒ at line 55), because they are equivalent to each other in our case. Note that we also change variable names. For instance, `$nextState` (❷ at line 24) is changed to `$nState` (Ⓑ at line 55).

7.2 Simplified Automaton Example

Fig. 10 shows a simplified version of real OSM's automaton consisting of 727 nodes and 3788 edges. The real instance of MASTERMIND delivers a simple webshell as malicious payload via benign websites USA Today [8], Weather in New York [97], Trinity Church in Boston [7], Houston Rockets on Twitter [5] and ESPN [4]. The automaton consists of 144 nodes and 766 edges. All nodes are placed to form a large outer circle. Edges representing transitions between the nodes. Inputs and outputs are omitted due to the space. We also highlight nodes and edges relevant to malicious payload generation using red colors. In this simplified example, there are 29 red nodes and 30 red edges out of 115 and 736 black nodes and edges respectively. The real malware contains the same number of red nodes and edges while it has significantly more number of black nodes and edges (698 nodes and 3022 edges). Note that such information (red nodes and edges for malicious payload delivery) is only for representation (hence is not available to analysts). Note that even the simplified version clearly shows that it is highly complex (with a number of nodes and edges), showing that it is challenging to reverse-engineer.

```php
private function _parseFormula($formula, Cell $pCell = null)
{
  ...
  while (true) {
    $opCharacter = $formula[$index];
    if ((isset(self::$comparisonOperators[$opCharacter])) &&
        (strlen($formula) > $index) && ... ) {
        $opCharacter .= $formula[++$index];
    }
    $opCharacter = $seq[$index];
    $oldPath = $path;
    foreach ($currentStates as $i=>&$state)
      $stateNo[$i] = array_search($opCharacter, $matrix[$state][0]);1
    $isOperandOrFunction = preg_match($regexpMatchString,
                                substr($formula, $index), $match);
    if ($opCharacter == '-' && !$expectingOperator) {
        $stack->push('Unary Operator', '~');
        ++$index;
    } else {
        ...
    }
    foreach ($stateNo as $value) {
      $tempNextStates = $matrix[$state][1][$value];
    2 $nextStates = array_merge($nextStates, $tempNextStates);3
    }
    ...
```

(a) Variant 1

```php
private function _parseFormula($formula, Cell $pCell = null)
{
  ...
  while (true) {
    $opCharacter = $seq[$index];
    $oldPath = $path;
    foreach ($currentStates as $i=>&$state)
      $stateNo[$i] = array_keys($matrix[$state][0], $opCharacter)[0];A
    $opCharacter = $formula[$index];
    if ((isset(self::$comparisonOperators[$opCharacter])) &&
        (strlen($formula) > $index) && ... ) {
        $opCharacter .= $formula[++$index];
    }
    for ($i = 0; $i < count($stateNo); $i++)
      $tempNextStates[$i] = $matrix[$state][1][$stateNo[$i]];
    $isOperandOrFunction = preg_match($regexpMatchString,
                                substr($formula, $index), $match);
    if ($opCharacter == '-' && !$expectingOperator) {
        $stack->push('Unary Operator', '~');
        ++$index;
    } else {
        ...
    }
    foreach ($tempNextStates as $i=>$value)
    B $nStates[$i] = $nStates[$i] + $value; C
    ...
```

(b) Variant 2

Figure 9: Randomized Code Snippets

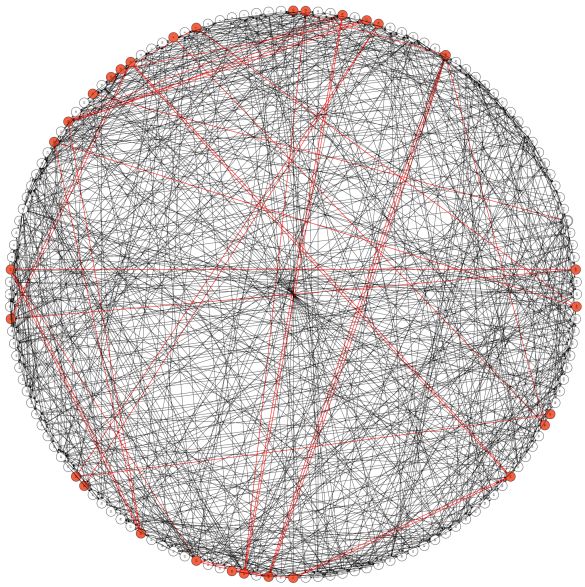


Figure 10: Example automaton (Simplified – approximately 20% of the original). Nodes are placed on the outer circle. Red nodes and edges are for the malicious payload generation.