Identifying Behavior Dispatchers for Malware Analysis

Kyuhong Park
Georgia Institute of Technology
kpark302@gatech.edu

Burak Sahin
Georgia Institute of Technology
buraksahin@gatech.edu

Yongheng Chen
Georgia Institute of Technology
ne0@gatech.edu

Jisheng Zhao
Georgia Institute of Technology
jisheng.zhao@cc.gatech.edu

Evan Downing
Georgia Institute of Technology
edowning3@gatech.edu

Hong Hu
Penn State University
honghu@psu.edu

Wenke Lee
Georgia Institute of Technology
wenke@cc.gatech.edu

Abstract

Malware is a major threat to modern computer systems. Malicious behaviors are hidden by a variety of techniques: code obfuscation, message encoding and encryption, etc. Countermeasures have been developed to thwart these techniques in order to expose malicious behaviors. However, these countermeasures rely heavily on identifying specific API calls, which has significant limitations as these calls can be misleading or hidden from the analyst. In this paper, we show that malicious programs share a key component which we call a *behavior dispatcher*, a code structure which is intercepted between various condition checks and malicious actions. By identifying these behavior dispatchers, a malware analysis can be guided into behavior dispatchers and activate hidden malicious actions more easily.

We propose BDHunter, a system that automatically identifies behavior dispatchers to assist triggering malicious behaviors. BDHunter takes advantage of the observation that a dispatcher compares an input with a set of expected values to determine which malicious behaviors to execute next. We evaluate BDHunter on recent malware samples to identify behavior dispatchers and show that these dispatchers can help trigger more malicious behaviors (otherwise hidden). Our experimental results show that BDHunter identifies 77.4% of dispatchers within the top 20 candidates discovered. Furthermore, BDHunter-guided concolic execution successfully triggers 13.0× and 2.6× more malicious behaviors, compared to unguided symbolic and concolic execution, respectively. These demonstrate that BDHunter effectively identifies behavior dispatchers, which are useful for exposing malicious behaviors.

CCS CONCEPTS

• **Security and privacy** → Intrusion/anomaly detection and malware mitigation; Malware and its mitigation.

KEYWORDS

Malware Analysis; Trigger-based Malicious Behaviors; Identification of Behavior Dispatcher; Program Analysis

ACM Reference Format:
Kyuhong Park, Burak Sahin, Yongheng Cheng, Jisheng Zhao, Evan Downing, Hong Hu, and Wenke Lee. 2021. Identifying Behavior Dispatchers for Malware Analysis. *In Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21), June 7–11, 2021, Hong Kong, Hong Kong.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3433210.3457894

1 Introduction

Malware has been a major threat to computer systems for decades, consistently jeopardizing millions of victims [22]. Even worse, companies can receive up to 350,000 malware samples every day [3]. To handle such a large volume of samples, automated dynamic malware analysis systems tend to be a popular solution. However, it is challenging to unveil malicious behaviors automatically because malware often contains behaviors that are activated only when appropriate triggering conditions are satisfied, called *trigger-based behaviors* [11, 43]. The trigger-based behaviors tend to come after a wide spectrum of evasion techniques, such as interacting with a command-and-control (C2) server and probing its execution environment. These techniques usually involve common computational tasks, which are difficult to analyze or bypass automatically, like encryption and checksums. In this paper, we collectively call these techniques and tasks *checks*.

Prior works attempt to detect and mitigate various types of complex checks [12, 23, 24] or fulfill triggering conditions by using API call-guided techniques [11, 28] atop fuzzing [8, 44] or symbolic execution [14, 36]. These approaches start from a predefined set of suspicious API calls and identify their invocations by statically dissecting a malware binary and dynamically executing the binary. Those invocation sites will then serve as entry-points to facilitate further dynamic analyses, such as fuzzing and concolic execution, for triggering malicious behaviors. However, these approaches have two limitations. First, malware commonly adopts evasion techniques to hide invocations of suspicious API calls [9]. Thus, only a small portion of API calls will be identified due to low code coverage [15]. Second, complex checks render the above approaches difficult in practice because there can be complicated checks between malicious actions and invoked API calls. Bypassing complex checks is difficult for prior solutions, thus the malicious actions can be unreachable [5, 20]. For example, malware `L0rdix RAT` uses `AES` to decrypt its C2 messages and verifies their integrity using `sha256`. API call-guided approaches typically fail to pass this check and thus miss malicious behaviors.

Given these limitations, we examine malware source code and malware samples from a public malware database [29] to understand how malicious actions are triggered. Our key observation is that the trigger-based malicious actions are usually grouped together and are *immediately preceded by various path selection operations*. We call these operations a *behavior dispatcher*, as it is the last roadblock before reaching the trigger-based behaviors. For instance, a bot often ships with a command dispatcher that receives various commands from the C2 server, compares the command with the expected values, and triggers the appropriate behaviors. These three actions denote the pattern to identify a behavior dispatcher. Further, we generalize the behavior dispatchers as a common control-flow pattern that can be identified via pattern matching on a control-flow graph (CFG): ❶ it contains a set of conditional branch operations that are defined by branch conditions, ❷ it directs program execution to a malicious action once the condition is satisfied, and ❸ if the match fails, it proceeds to the next check.

In this paper, we introduce BDHunter, a system that effectively identifies behavior dispatchers in trigger-based malware. We demonstrate that guiding a program execution to behavior dispatchers in a malware analysis system can help unveil trigger-based malicious behaviors. BDHunter uses two algorithms to detect behavior dispatchers: one based on control-flow pattern matching and another based on weighted API calls. In the pattern-based approach, BDHunter constructs the CFG from a malware sample and performs intra-procedural dataflow analysis to identify code blocks matching the dispatcher pattern. In the weight-based approach, BDHunter assigns each API call an initial weight based on its importance for constructing malicious actions and aggregates these weights to the callers of these API calls. If the suspicious API calls can be collected while dynamically executing a malware sample, BDHunter utilizes both of these approaches to produce a set of candidate behavior dispatchers although the weight-based approach is mainly used to enhance the pattern-based approach. Otherwise, BDHunter uses the pattern-based approach to generate a set of candidate behavior dispatchers.

Overall, we evaluate BDHunter's performance in ❶ identification of behavior dispatchers and verification of the results on 302 real-world malware samples chosen from VirusTotal [40]; ❷ usefulness of the identified behavior dispatchers in concolic execution (as an application of BDHunter), which can be used to guide the execution to trigger malicious behaviors; ❸ robustness against adversarial attacks that target to evade BDHunter. Our experimental results show that BDHunter is effective and efficient in behavior dispatcher identification as well as revealing trigger-based malicious behaviors with BDHunter-guided concolic execution. Furthermore, we show that BDHunter is robust against adversarial attacks that attempt bypass identification of behavior dispatchers.

In summary, we make the following contributions:

- We propose a practical approach to identify *behavior dispatchers*, the last roadblock before reaching malicious behaviors.
- We prototype our approach as BDHunter[1], a system that automatically detects behavior dispatchers from malware.
- We demonstrate the application of behavior dispatchers in helping expose hidden malicious behaviors.
- We evaluate our tool on 302 recent real-world malware samples.

2 Problem and Approach

In this section, we describe our motivation for detecting behavior dispatchers in trigger-based malware and discuss limitations of existing solutions, which fail to handle complicated checks. Then, we show how behavior dispatchers help reveal stealthy actions.

2.1 Motivating Example

Figure 1 shows the source code and CFG of a malware sample (simplified) from the `Dexter` malware family, which contains trigger-based malicious behaviors as well as complicated checks. In Figure 1b, each rectangle represents one basic block, and its labels correspond to the line numbers in the source code in Figure 1a. Starting from the function entry `HttpMain`, the execution retrieves the private key (line 4) and the OS version (line 5), and compares the version to the expected value (line 6) at block `L4`. If the version matches, it goes to block `L7` to receive a command from the C2 server (line 7). If not, the malware will immediately exit at block `L6` to avoid introducing observable malicious behaviors. After decrypting and verifying the command via `RSA` and `sha256` at block `L8` (line 8), the execution walks through a sequence of basic blocks `L9`, `L11`, and `13` (line 9, line 11, and line 13, respectively), each comparing the command (register `edx` in assembly code and `pCMD` in source code) with a trigger value. If they match, the malware invokes the corresponding function to complete the command: e.g., updating the address of the C2 server at block `L10` (line 10), checking the aliveness of the C2 server at block `L12` (line 12), or downloading new malware at block `L14` (line 14). Otherwise, it proceeds to the next check. Through the malware sample, we observe that its malicious behaviors include at least `Update`, `Checkin`, and `Download`. A failure to reveal any of these behaviors may construct a weak defense.

2.2 Limitations of Existing Methods

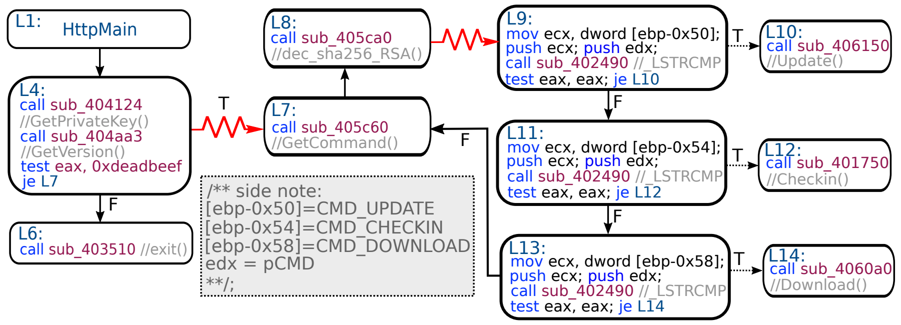
Existing works attempt to either mitigate evasive logic (e.g., detecting running environment) or fulfill triggering conditions (e.g., feeding proper inputs) atop symbolic execution or fuzzing to unfold malicious behaviors. However, each approach cannot easily trigger malicious behaviors from malware [5, 6, 32]. The reason is that malware tends to include ❶ a combination of multiple evasion techniques and ❷ complex computational tasks, including the encryption/decryption of data and the computation of hashes. We call these *checks* in malicious code. Unless all complicated checks are satisfied and mitigated one by one, the analysis techniques cannot reach the location of conditions that trigger malicious actions.

For example, at line 6 of Figure 1a, the malware requires the OS version to be a specific 32-bit value. Fuzzing techniques are well-known to be inefficient in solving these complicated checks [20]. Therefore, fuzzing-based explorations will likely get stuck on path

[1]The source code of BDHunter can be found at https://github.com/kp2bit/bdhunter

```c
1  void HttpMain() {
2    char Commands[255], Url[255];
3    char *pCMD = &Commands;
4    char *pKey = GetPrivateKey();
5    DWORD dwVersion = GetVersion();
6    if (dwVersion != 0xDEADBEEF) exit();
7    while (GetCommand(Url, pCMD) != 0) {
8      dec_sha256_RSA(&pCMD, &pKey);
9      if (!_LSTRCMP(pCMD, CMD_UPDATE))
10       Update(Url);
11     else if (!_LSTRCMP(pCMD, CMD_CHECKIN))
12       Checkin();
13     else if (!_LSTRCMP(pCMD, CMD_DOWNLOAD))
14       Download(Url);
15     ...
16 }}
```

(a) Simplified malware source code



```
L1:  HttpMain

L4:
call sub_404124
//GetPrivateKey()
call sub_404aa3
//GetVersion()
test eax, 0xdeadbeef
je L7

L6:
call sub_403510 //exit()

L7:
call sub_405c60
//GetCommand()

L8:
call sub_405ca0
//dec_sha256_RSA()

L9:
mov ecx, dword [ebp-0x50];
push ecx; push edx;
call sub_402490 //_LSTRCMP
test eax, eax; je L10

L10:
call sub_406150
//Update()

L11:
mov ecx, dword [ebp-0x54];
push ecx; push edx;
call sub_402490 //_LSTRCMP
test eax, eax; je L12

L12:
call sub_401750
//Checkin()

L13:
mov ecx, dword [ebp-0x58];
push ecx; push edx;
call sub_402490 //_LSTRCMP
test eax, eax; je L14

L14:
call sub_4060a0
//Download()

/** side note:
[ebp-0x50]=CMD_UPDATE
[ebp-0x54]=CMD_CHECKIN
[ebp-0x58]=CMD_DOWNLOAD
edx = pCMD
**/;
```

(b) Simplified malware control-flow graph

Figure 1: Source code and control-flow graph of a simplified malware. This malware accepts commands from its C2 server, decrypts each command through the private key and sha256, and dispatches the execution accordingly. It only works on a particular Windows version (e.g., `0xDEADBEEF`). In the control-flow graph, the red polyline indicates a condition that is hard to satisfy by program analysis techniques.

`HttpMain`→`L4`→`L6` and will not trigger branch `L4`→`L7` (the red polyline in Figure 1b). Symbolic execution is good at solving these constraints of value comparisons [5] and thus can trigger branch `L4`→`L7`. However, symbolic execution can have large computation and memory costs due to path explosion, and thus cannot efficiently handle computation-intensive calculations, such as *checks* [5]. Although [12] tries to overcome encoding and encryption-related challenges by introducing a decomposing and re-stitching technique, this technique is required to identify the inverse functions of encryption and encoding functions to complete re-stitching. However, identifying the inverse function is difficult if malware authors use a customized encryption function [42]. Worse, an inverse function may not exist if the malware uses asymmetric encryption (e.g, `RSA`). At line 8 of Figure 1a, the malware decrypts and verifies the command with `RSA` and `sha256`, which will prevent existing symbolic execution tools [5] from triggering condition block `L8` (another polyline in Figure 1b). The computation and memory overheads make these methods infeasible to analyze real-world malware that have significantly more checks than this example.

Existing approaches rely on identifying a set of suspicious API calls, which are predefined by security analysts based on their domain expertise. For example, analysts treat `HttpOpenRequestA` as a suspicious API call, as it may enable malware to communicate with its C2 server. These approaches will perform traditional analysis techniques starting from where the API calls return. In Figure 1a, function `GetCommand` receives a command by invoking network API calls `HttpSendRequest` and `InternetGetCookie` (not shown in the figure). Existing techniques will identify these two suspicious API calls and begin analysis after returning from `GetCommand`.

Unfortunately, API call-guided approaches have two major limitations. First, they require the malware to *explicitly* invoke predefined API calls, such as `HttpSendRequest` to receive network packets. However, malware tends to hide them by obfuscating API call names [9]. For example, malware `Carbanak` stores encrypted API call names in its binary and will dynamically decrypt the name and resolve the API call address by calling `GetProcAddress`. Thus, static malware analysis cannot be guaranteed to find any meaningful API call names. Although dynamic execution can resolve the obfuscated API calls, it can only observe the ones invoked during execution, which can suffer from low code coverage [15]. Second, even if we discover suspicious API calls, the malware may still contain a large number of complicated checks between more suspicious API invocations and launching its actions. For example, the API call-guided exploration will help the analysis skip branch `L4->L7`, but there is another roadblock, `L8->L9`, concealing more malicious actions.

2.3 Hunting Behavior Dispatchers

Although the advantage of API call-guided techniques comes from their ability to skip complicated checks (i.e, skipping branch `L4`→`L7`), API call-guided methods only skip checks between the malware entry point and suspicious API calls. Conservativeness is desired here because if we skip too many checks, our analysis may miss behaviors that are only reachable from anterior locations. In the ideal case, instead of focusing on mitigating the combinations of complicated *checks* one-by-one, we would fast-forward our analysis to the location that can reach all malicious behaviors.

Through manual analyses on 21 malware source codes and 31 malware samples (§A.3), we identified a generic component that can help skip more checks and still reach conditions that trigger malicious actions. We call this component *behavior dispatcher*. A behavior dispatcher contains many conditional branch operations (a set of instructions that direct the program execution regarding the branch condition). Each of those operations checks a common variable with a value that satisfies the condition that triggers a malicious action, called a *trigger value*. If the condition is satisfied, the execution is directed to the branch that triggers the malicious action. Otherwise, the execution proceeds to the next conditional branch operation. This process continues until all conditions fail.

Here we give the formal definition of behavior dispatchers and its relevant terms used in this paper. Behavior dispatchers are composed of a set of execution paths obtained from malware. Each execution path (denoted by p) in the behavior dispatchers is presented as a sequence of basic blocks. See the definition below:

Definition 2.1 (Conditional Branch Operation). a code section that justifies the branch target selection, denoted by $CondBrOp$, located at a basic block, presented as a comparison operation $CmpOp$ that affects branch condition, and a branch instruction $BrInst$ that directs the execution to the satisfied path.

Definition 2.2 (Common Variable). a program identifier, denoted by $CommVar$, presented as a variable that is used by multiple

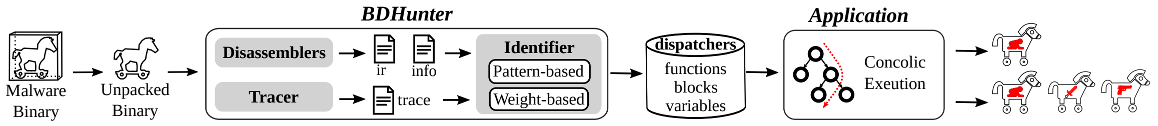


Figure 2: Overview of BDHunter and its application. Our system takes unpacked malware binaries as input. It collects both static information (such as CFGs) with a disassembler and dynamic information (such as the invoked API calls) using a tracer. It then feeds the static information into the pattern-based identifier and the dynamic information into the weight-based identifier when the suspicious API calls can be collected. The two identifiers generate a list of candidate behavior dispatchers. Concolic execution can utilize the dispatcher information to expose malicious behaviors in malware samples.

*CondBrOp*s in a path p for justifying their following branch target selection.

Definition 2.3 (Behavior Dispatchers). behavior dispatchers are denoted by S, presented as a set of paths: $p_0, p_1, ... p_n$. Each path p_i is composed with a set of basic blocks that contain either *CondBrOp* or *CommVar*.

In practice, a conditional branch operation presents the code pattern for the trigger-based malicious behavior in malware samples (i.e., its branch condition presents the comparison between the input variable and a trigger value). To get a better understanding of the definition of behavior dispatcher, we use Figure 1 as an example. In Figure 1, lines 9, 11, 13 form a behavior dispatcher, which contains three sequential conditional branch operations (blocks `L9`, `L11`, and `L13` in Figure 1b) that compare the common value `pCMD` (i.e., [edx] in assembly code) with the trigger values `CMD_UPDATE`, `CMD_CHECKIN`, `CMD_DOWNLOAD`, respectively (i.e., [ebp-0x50], [ebp-0x54], [ebp-0x58] in assembly code). The comparison operation in the figure is performed by calling `_LSTRCMP`, or the call instruction to `sub_402490` in assembly code, and the return value is used in a comparison instruction (i.e., the `test` instruction) that affects branch condition. The branch instruction (i.e., the `je` instruction) directs the program execution to the satisfied path. Once a condition matches, the code will run one malicious action. Starting from the dispatcher, we can now perform program analysis techniques, as many roadblocks have been skipped. In the example above, starting from block `L9`, both fuzzing and symbolic execution can explore paths easily.

As shown in Table A.1, BDHunter can identify seven categories of behavior dispatchers that are related to C2 commands, file systems, processes, network, registry, time, and system environment. C2-related dispatchers usually wait for a list of commands from the C2 server and launch specified actions. A file system-related and registry-related dispatcher checks the status of some particular files or registries, such as the existence of a log file indicating the system has been compromised. Network-related dispatcher are activated when the malware send/receive specific packets. A time-related dispatcher triggers malicious actions only at a particular time. An environment-related dispatcher might only unfold a malicious behavior if the underlying system matches the specific version, like the code in Figure 1. Our experiments show that if malware contains at least one of the seven behavior dispatchers, BDHunter is able to identify them within the malicious binary.

3 BDHunter

Figure 2 shows the pipeline of BDHunter. The goal of BDHunter is to automatically identify behavior dispatchers in trigger-based malware. BDHunter takes an unpacked malware binary as input and outputs a list of candidate behavior dispatchers, which includes the candidate's related functions, basic blocks, and variables. The behavior dispatchers can then assist program analysis techniques, such as concolic execution, to trigger hidden malicious actions. BDHunter takes two steps to identify dispatchers from a given malware sample. First, it analyzes malware program structures and collects relevant information via static analysis and dynamic execution (§3.1). Second, with the extracted dynamic and static information, BDHunter applies two heuristic-based algorithms to identify behavior dispatchers: one is based on the unique code pattern of behavior dispatchers (§3.2), while another relies on a weighting scheme to score each function (§3.3).

3.1 Malware Information Collection

The first step of identifying behavior dispatchers is to collect preliminary information from a malware sample, including invoked API calls from dynamic execution and control-flow graphs from static analyses. BDHunter uses a dynamic analysis environment to collect runtime information and to record the invoked API calls. Our recording does not seek to obtain a complete set of suspicious API calls, but simply collects as many as possible to improve the weight-based method. During the static analysis, BDHunter utilizes a disassembler to lift the malware binary into an intermediate representation (IR). The IR analysis performs four tasks: ❶ extracting auxiliary information, including symbols, import and export tables; ❷ detecting and removing irrelevant functions that are borrowed or inlined from well-known libraries and thunks; ❸ identifying variables (e.g, local, stack, global) and basic blocks that are related to conditionals, load & store, and comparisons to identify *CondBrOp*s and *CommVar*s; ❹ constructing control-flow and call graphs by leveraging IR and execution traces.

BDHunter leverages BinaryNinja [39], an industry-standard disassembler, and its IR, called BNIL. BNIL provides a more abstract semantic representation of the assembly instructions (e.g., x86 and x86-64) by mapping assembly instructions to IR instructions [39]. With BNIL, BDHunter can efficiently identify instructions that are related to conditionals, load & store, and comparisons.

3.2 Pattern-based Identification

The pattern-based method statically searches within the malware binary to identify basic blocks with dispatcher patterns: a behavior dispatcher contains a set of *CondBrOp*s, each which compares the *CommVar* with a trigger value.

Algorithm 1 details the algorithm of our pattern-based identification. Function `PatternBasedIdentifier` takes the CFG of each function as input and outputs a list of candidate behavior dispatchers. It initializes several local variables and then invokes `CheckOneBlock` (line 4). `S` contains identified behavior dispatchers; `path` is an execution path that contains two data structures: one is a list of all blocks in the current path, and another is a set of blocks that have *CondBrOp*s, called `xBBs` in the path. `CheckOneBlock` examines the current block and recursively handles all successor blocks.

First, it checks whether the current `BB` contains *CondBrOp* (line 7). Figure 1b show that a set of blocks (block `L9`, `L11`, and `13`) use a call instruction (i.e. the *CmpOp*) to compare the command (register `edx`, the *CommVar* in the set of call instructions) with a trigger value. The following comparison and conditional branch instructions, which are *CmpOp*s and *BrInst*s, will then use the call instruction's return value to justify which branch should take. Instead of relying on comparison-related API calls, BDHunter looks for the *CondBrOp* pattern that presents as the *CommVar* shared by a set of *CmpOp*s (i.e. either call instructions or comparison instructions) that impacts the following branch target selection. This is because we cannot guarantee to identify comparison-related API (e.g., `STRCMP`) calls due to embedded string obfuscation or customized comparison functions. In the example, although the malware source code uses a well-known comparison function (e.g., `LSTRCMP`), Figure 1b shows the `LSTRCMP` as `sub_402490` due to the reasons above.

If the current `BB` contains *CondBrOp*, we add `BB` into the current path's basic block list `BBs` and its *CondBrOp* into `CondBrs` set (line 8-9). Otherwise, only `BB` is added into `BBs` (line 11). If the current path contains more than `WSize` blocks — the minimum size of a behavior dispatcher— we check whether it contains enough blocks with *CondBrOp*s (line 13) and whether all blocks with conditions share some *CommVar*s (line 14). Here, `WSize` is the number of basic blocks inside the path, while K_{cmp} is a threshold defined by our observation of the 52 malware samples that we studied (see details in §3.2.2). If so, we merge the current `path` into `S`, the set of candidate behavior dispatchers (line 15). Note that `merge` is not simply inserting `path` into `S`. Instead, we merge two `paths` via a shared *CommVar*, denoted as p_r = merge(p_i, p_j, *CommVar*). If both p_i and p_j share a *CommVar* in some of their *CondBrOp*s, then p_i, p_j are merged into a single path p_r that preserves the *CondBrOp*s that uses the *CommVar*. For the uncommon operations between those *CondBrOps*, the longer path is selected. In this way, we can prevent one long dispatcher from being split into several smaller chunks.

After checking the current block `BB`, `CheckOneBlock` removes the header of `path`'s `BBs` and its relevant *CondBrOp*s from `xBBs` to reduce the complexity (line 16-17) and invokes itself to recursively handle all successors one by one (line 18-19). Note that we duplicate `path` for the recursive call to guarantee a new path is expanded, as `CheckOneBlock` may update them, as in line 8 and line 16. By applying `PattenBasedIdentifier` on all functions in the malware sample, the pattern-based method can find candidate behavior dispatchers. All of those identified candidates are sorted by their total number of basic blocks and the ratio of *CondBrOp*s. This assigns higher priority to those candidates with more basic blocks over ones with fewer. In the case that two candidates have the same number of basic blocks, the one which has more *CondBrOp*s is considered for assigning higher priority.

3.2.1 Behavior Dispatchers via Tight Loops. Figure A.2 shows the source code and the CFG of a `Zeus` malware sample which has a different dispatcher pattern compared to the one we discussed before.

Algorithm 1: Pattern-based Dispatcher Identification

```
Input  : CFG :- control-flow graph of a function
Output : S :- the set of candidate behavior dispatchers
1  Procedure PatternBasedIdentifier(CFG)
2      S = ∅; path.BBs = ∅; path.CondBrs = ∅;
3      BB = getEntryBlock(CFG);
4      CheckOneBlock(S, CFG, BB, dup(path));
5      return S;
6  Procedure CheckOneBlock(S, CFG, BB, path)
7      if hasCondBr(BB) then
8          path.BBs.append(BB);
9          path.CondBrs.append(getCondBr(BB));
10     else
11         path.BBs.append(BB);
12     if path.BBs.length() > WSize then
13         if path.CondBrs.length() > $K_{cmp}$ * WSize then
14             if hasCommVar(path.CondBrs) then
15                 S.merge(path)
16         topBB = path.BBspopFront();
17         path.CondBrs -= getCondBr(topBB);
18     foreach succBB ∈ getSuccessors(BB) do
19         CheckOneBlock(S, CFG, succBB, dup(path))
20     return;
```

Instead of having many *CondBrOp*s in the code, it implements a compare-dispatch routine with one compare instruction and one indirect function call inside of a loop. In Figure 4a, for each iteration of the loop (lines 5-9), it compares the command `cmd` with one element (a trigger value) of the array `cmds` (line 7). If the condition is satisfied, `Zeus` indirectly jumps to the function that implements the malicious actions (line 8), like `execute` for running arbitrary commands. Thus, we adjust Algorithm 1 to detect dispatchers with such a pattern. In this case, the pattern-based method first analyzes the given CFG to identify loops that contain comparison instructions and indirect jumps and calls as *CmpOp*s. Then, the method calculates the percentage of *CmpOp*s and indirect jumps and calls for each loop. If the percentage is higher than a threshold K_{loop}, then we treat the loop as a candidate behavior dispatcher (details in §3.2.2). The variable used in the *CondBrOp*s is the triggering condition, while the indirect call is used to launch actions.

3.2.2 Sliding Window and Threshold. Algorithm 1 contains three user-defined thresholds to detect behavior dispatchers. We choose these values through the analysis of the 21 source codes and 31 malware samples (§A.3) from [29]. The first threshold, `WSize`, defines the minimum number of basic blocks inside of a behavior dispatcher. A larger `WSize` requires more blocks to have *CondBrOp*s and thus may miss some dispatchers with few operations (i.e., false negatives), while a smaller `WSize` may introduce too many false positives. The second threshold, K_{cmp}, defines the minimum percentage of blocks with *CmpOps* inside a behavior dispatcher. The last threshold, K_{loop}, defines the minimum percentage of blocks with *CondBrOp*s inside the loop for identifying loop-based dispatchers. Through experiments using different threshold values among the 52 malware samples, we found that the identified dispatchers usually have at least three basic blocks with 65% of comparison operations to give the optimal results. Therefore, we set `WSize` to 3, K_{cmp} to 65%, and K_{loop} to 70% for our evaluation (§5) because they result in the lowest false positives (FPs) and false negatives (FNs).

Although none of the 52 samples are used in our evaluation, their families make up 4.7% of our evaluation dataset (details in §A.4).

3.3 Weight-based Identification

BDHunter collects the invoked API calls during dynamic execution to use for its weight-based identifier. It uses these calls to help identify behavior dispatchers to improve efficiency and accuracy on top of the pattern-based approach. The intuition behind the weight-based identifier is that ❶ a behavior dispatcher can reach many malicious actions, ❷ each of its callees can only reach a part of these actions, and ❸ each of its callers can reach the same number of malicious actions. This intuition works for both patterns of behavior dispatchers, as shown in Figure 1 and Figure A.2. Once the code blocks in a function satisfy these three criteria, we treat the function as a candidate function that contains behavior dispatchers.

The identification of suspicious API calls is important in the weight-based identifier. We utilize a list of API calls (§A.5) from a report of a security vendor [17], and the weight for each suspicious API call is based on its importance for construction attacks. For example, if the malware sample is known to use HTTP-based C2 communication, a security analyst may assign relatively high weights on HTTP-related API calls, and low weights on comparing-related API calls for detecting C2-related dispatchers. Additionally, users of BDHunter can customize their own set of suspicious API calls and the weights for each API calls based on their experiences or their target-specific attacks. We describe the limitations of the weight-based identifier in §5.1.2.

Algorithm 2 details our algorithm. It takes the CFGs of the whole program and the weight for each suspicious API call as input. First, the algorithm iteratively propagates the weight from API calls or functions to their callers (lines 2-13). Specifically, if all of the callees of one function have a weight, the algorithm takes the sum of all weights as the weight of the current function (lines 10 and 12). The weight-based method keeps propagating until it cannot make any new updates to any function weight. Second, (lines 14-22), the method calculates the difference of weights between callers and callees. `diff_callees` is the average weight difference between each function and its callees, which should be a large value for the dispatcher based on our intuition ❷. `diff_callers` is the average weight difference between each function and its callers, which should be a small number for the dispatcher based on our intuition ❸. `weight_diff` is the difference between `diff_callees` and `diff_callers` and should be a large value for the dispatcher according to intuitions ❷ and ❸. Therefore, if the `weight_diff` is larger than a threshold, the method treats the current function as a candidate behavior dispatcher. The method sorts the set of candidates based on `weight_diff` and output the result.

3.4 Implementation

BDHunter is implemented in 3,452 lines of Python 3 code, which utilizes BinaryNinja and BNIL. Further, BDHunter uses IDA Pro FLIRT [19] to remove irrelevant functions such as well-known libraries and thunk functions, which optimizes our methods for better performance. The system leverages DynamoRIO [10] to trace the malware's dynamic execution. The current prototype works on x86 PE binaries. However, it can be easily extended to other architectures because DynamoRIO supports cross-platform and BNIL is an architecture-independent IR of machine code and provides a more abstract semantic than assembly codes (e.g., x86 and x86-64).

4 Revealing Malicious Behaviors

To demonstrate the utility of behavior dispatchers, we apply them to a concolic execution engine to activate trigger-based malicious behaviors for malware analysis. Fuzzing [8, 44] and concolic execution [14, 36] techniques have been applied to benign applications. Typically, concolic execution requires manual effort to locate the symbolic source to hit certain targets for discovering vulnerabilities and new program states. However, performing this manual analysis in malware can be a tedious and time-consuming job due to evasion techniques and complicated checks. Moreover, limitations of existing methods for triggering behaviors discussed in §2.2. Concolic execution in malware analysis would be more productive in triggering behaviors if it could automatically identify the target location(s). This would reduce the manual effort required to locate the symbolic source to hit the target and overcome existing limitations. The location of a dispatcher can be served as the target location to reveal malicious actions in trigger-based malware. Hence, concolic execution can leverage the dispatchers to circumvent either unnecessary or complicated checks to activate trigger-based behaviors.

We developed a custom concolic execution engine to show the practicality of identifying behavior dispatchers. Our engine is built on top of DynamoRIO and angr [36], and leverages BDHunter to direct the execution towards behavior dispatchers. DynamoRIO runs on Windows 7 in a virtual machine as Concrete Execution Engine (CEE) and angr runs outside the virtual machine as Symbolic Execution Engine (SEE). They communicate through gRPC [37].

First, we run the malware in CEE to extract the execution trace. Second, SEE parses the execution trace and finds the basic block closest to behavior dispatcher. To determine and prioritize the closest block to behavior dispatcher, we adopt the path selection algorithm described in AFLGo [8]. SEE computes the distance between the executed blocks and the dispatcher to find the closest block. Then, SEE starts its exploration and prioritizes branches closer to the dispatcher. At this point, SEE and CEE work simultaneously. SEE sends a breakpoint list to CEE which includes the dispatcher, the function that contains the dispatcher, and the address of the BB closest to the dispatcher. SEE manages the concrete state in CEE via updating the memory value, register value, or forcing the conditional branch in case the constraint solver fails. CEE and SEE iteratively perform the above process until they reach the dispatcher. Once the function that contains the dispatcher is reached, SEE symbolizes all the function arguments and attempts to visit the dispatcher. When a dispatcher is visited, SEE computes the triggering conditions for that behavior and sends the memory update command with a trigger value to CEE. When CEE receives the trigger value and register information, it makes necessary updates and resumes the concrete execution. With the steps above, the concolic engine tries to find the triggering conditions in the dispatcher. In summary, the SEE interleaves the concrete execution at the point of execution closest to the function of behavior dispatchers, and CEE performs the requests (reads concrete data, updates concrete data, etc.) of SEE.

Though our solution may not identify all triggering conditions and explore all malicious behaviors, it can extract information (e.g., the dispatcher and its latent behaviors) without introducing many irrelevant paths or states. The engine outputs various information such as C2 commands, network activities, and environmental conditions. We can leverage them to apply more intensive concolic execution techniques by mixing dynamically discovered features with statically available information. In §5.4, we show the results of BDHunter-guided concolic execution.

5 Evaluation

We evaluate BDHunter on 302 real-world malware samples to demonstrate its effectiveness for detecting behavior dispatchers and helping trigger malicious actions. Our evaluation aims to answer the following questions:

- Can BDHunter accurately identify behavior dispatchers from real-world malware samples (§5.1)?
- Is BDHunter scalable to handle a large number of malware samples (§5.2)?
- Is BDHunter resilient against obfuscation techniques to attempt to evade the identification of behavior dispatchers (§5.3)?
- Are behavior dispatchers useful for an application (e.g., concolic execution) to reveal more malicious behaviors (§5.4)?

Dataset. For the evaluation of BDHunter, we collected 7,147 malware samples from VirusTotal [40]. We then unpacked our samples using unipacker [27]. Although it is not as complete as prior works [13, 16], the tool produces a valid executable if it is successful. Since unipacker can unpack most of the popular packers used by malware [38] and is an open-sourced tool, we chose unipacker to unpack our dataset. Unipacker was able to unpack 4,131 of the 7,147 samples. However, after unpacking, we noticed that a few samples had invalid PE formats (e.g., invalid sections and start address). After filtering them out (188 PE files), we apply BDHunter on the remaining 3,943 malware samples.

Experiment Setup. We use the pattern-based identifier to detect the seven types of behavior dispatchers with the three thresholds described in §3.2.2. To measure the weight-based method's effectiveness, we utilize the list of API calls (described in §3.3) for identifying C2-related dispatchers only. Due to their complexity, we did not attempt to identify other behaviors because of the potential number of API calls involved [41], although conceptually they can be explored in future work. Besides, C2 activities are highly related to trigger-based behaviors. We performed the experiments on a machine installed with 64-bit Ubuntu 16.04 and equipped with Intel Xeon Gold 6140 CPU and 500 GB RAM.

5.1 Accuracy of Dispatcher Identification

Ground Truth. Validating BDHunter requires checking ❶ the existence of trigger-based behaviors in a malware sample and ❷ that the behavior dispatcher denotes the last roadblock to trigger malicious behaviors. Since there is no publicly available ground-truth for us to verify the result and evaluate BDHunter, we have to build our own ground-truth. Since validating BDHunter requires manual reverse engineering on each malware, we randomly selected 302 samples (out of the 3,943 samples) from 127 distinct malware families labeled by AVClass [33] as shown in Table 1. We manually analyzed those samples to validate our results and leveraged malware analysis reports from a security vendor [17].

Label	#	Label	#	Label	#	Label	#
kolabc	49	virut	34	upatre	11	zbot	7
chir	6	crytex	6	scar	6	others	183
						Total	**302**

Table 1: Malware samples and their families. We apply BDHunter on 3,943 malware samples to find behavior dispatchers and analyze the results of 302 samples from 127 families. §A.4 provides the list of families.

Prioritization and False Positives. To measure the accuracy of identifying dispatchers, we used BDHunter to identify, at most, the longest 100 candidate behavior dispatchers per sample (BDHunter sorts the candidates by their total number of basic blocks as described in §3.2). We chose to limit the number of candidates due to the shear volume and because ❶ complicated malware can have many malicious behaviors triggered by many conditions [18], and ❷ some samples have a large number of functions (e.g., we found 7,927 functions in a single sample). Furthermore, we demonstrate that this threshold does not artificially improve our results.

For each reported candidate from 302 malware samples, our manual analysis validated that the pattern-based method detects at least one behavior dispatcher (from each of the seven types) from 280 samples. The weight-based method identifies C2 dispatchers from 102 samples. However, we identified 10 FPs and 12 FNs within 22 samples. We carefully inspected these 22 cases. For the 10 FPs, 4 out of the 10 samples do not utilize a dispatcher with conditional branch operations to trigger malicious behaviors. The other 6 samples cause a failure due to improper disassemble. For the 12 FNs, the samples are still either packed or obfuscated. In this case, the complex form of CFGs with a large number of basic blocks and edges can cause FNs in BDHunter. Overall, Table 2 shows that BDHunter detects each of the seven types of behavior dispatcher in 302 malware samples. On average, 87.2% of dispatchers in all the seven types are identified within the top 100 candidates (meaning that 12.8% of dispatchers are located outside of these top 100 candidates). At most 10% of dispatchers for each type are identified outside of the top 100 candidates except for time-related dispatchers. We discuss potential improvements and the limitations of BDHunter in §6.

5.1.1 Pattern-based Identification. In this evaluation, we use the three thresholds described in §3.2.2. The pattern-based method detects at least one behavior dispatcher in 280 samples (out of the 302 samples). Figure 5a shows that, within the top 20 behavior dispatcher candidates, BDHunter finds at least one dispatcher in 77.4% of the 280 malware samples. Within the top 100 candidates, BDHunter identifies at least one dispatcher within 90% of the 280 samples. Figure 5b shows that analyzing only 10% of the candidate functions in a sample reveals at least one dispatcher in around 90% of the samples. In summary, Figure A.7 shows that BDHunter greatly reduces the number of functions required for a malware analysis tool or an analyst to analyze in the samples. This, in turn, can reduce the amount of time and effort they would spend on manually identifying a dispatcher.

Per-type Accuracy. Table 2 shows the identified behavior dispatchers from all the seven types and the distribution of different types of behavior dispatchers. C2-related and file system-related

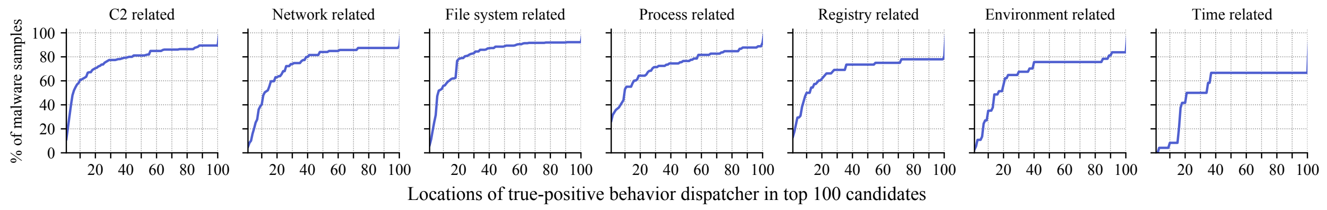


Figure 3: Distribution of samples with behavior dispatchers of different types, detected by the pattern-based identifier. The x-axis shows how many top candidates are considered, while the y-axis shows how many samples with behavior dispatchers have been detected.

Dispatcher Types	C2	File	Network	Process	Registry	Env	Time
# Dispatchers	265	347	119	98	68	37	24
# Samples w/ Dispatchers	202	180	85	81	58	29	18

Table 2: Behavior dispatcher distribution in 302 samples among different types, detected by the pattern-based identifier.

dispatchers are the most popular, while time-related dispatchers are uncommon in our dataset. Some samples contain multiple behavior dispatchers, which may fall into the same type. For example, BDHunter finds 265 C2-based dispatchers within 202 samples. We examined these samples further and found that for each command, the malware accepts sub-commands, where each sub-command dispatcher forms a behavior dispatcher and is detected by BDHunter.

Figure 3 shows the distribution of malware samples with different types of behavior dispatchers, as detected by BDHunter. For example, for C2, network, and file system dispatchers, their samples share similar distributions: in 74% of all samples, BDHunter identified a dispatcher within the top 30 candidates, while this percentage slowly increases to 90% when we consider the top 100 candidates. For process and registry dispatchers: in 70% of all samples, BDHunter identified a dispatcher within the top 30 candidates.

5.1.2 Weight-based Identification. Our weight-based identifier detects C2-related dispatchers in 102 out of the 302 malware samples. Since the pattern-based method identifies C2-dispatchers within 202 samples, this means that our weight-based method missed those dispatchers in 100 samples. We have categorized these FNs into three cases: ❶ usage of string obfuscation or dynamic loading to hide API calls, ❷ disconnected caller and callee due to indirect calls, and ❸ pre-defined suspicious API calls that are irrelevant in triggering malicious behaviors. For example, malware `Carbanak` uses complex string obfuscations with a combination of encryption and hashing to hide API calls. Figure 6a shows the distribution of 102 samples among the top 100 candidates. This figure shows that in over 78.4% of samples, BDHunter identified a dispatcher within the top 20 candidates and discovered dispatchers in almost all the samples when increasing to the top 100 candidates.

5.1.3 Comparison between the two methods. Figure 6b shows the comparison between the two methods for detecting C2 dispatchers among samples which contained API calls. The weight-based method can detect the behavior dispatchers within the top 20 candidates consistently across 102 samples, while the pattern-based method may introduce more FPs. As shown in the figure, the pattern-based method is more generic than the weight-based method, where the latter missed C2-dispatchers in 100 samples. The reason is that the pattern-based method relies on a common pattern within malware samples while the weight-based method relies on explicit API calls from the program. However, if suspicious API calls are explicitly called, the weight-based method can provide more accurate results. Thus, users of BDHunter should consider using both identification methods to complement each other and improve accuracy. Both methods can be configured to improve accuracy by modifying the thresholds and list of suspicious API calls.

5.2 Analysis Efficiency

We measured the performance of BDHunter on the task of detecting behavior dispatchers from 3,943 malware samples. BDHunter can complete the pattern-based identification in an average of 74.03 seconds and the weight-based identification in an average of 43.85 seconds per malware samples. Thus, BDHunter can handle most real-world malware samples efficiently and can be used to process daily-reported malware within a reasonable amount of time.

5.3 Robustness of BDHunter

Adversaries aware of the existence of BDHunter could apply targeted obfuscation techniques to hinder the capabilities of BDHunter. To measure the resilience of BDHunter against these techniques, we evaluate BDHunter on the 20 sample programs from an anti-analysis benchmark [7] by applying an obfuscation tool: Obfuscator-LLVM (*ollvm*) [21]. The programs trigger an action only if a condition is matched to test the resilience. This is similar to trigger-based behaviors in malware, which are activated only when appropriate triggering conditions are satisfied. To realistically mimic trigger-based behaviors in malware (and not to skew the original code in the benchmark), we only add more triggering conditions that are compared with the given input (e.g., each of them triggers a different action) in the programs. We use *ollvm* to compile the 20 samples with the following configurations: no obfuscation, instruction substitution, bogus control flow, and control-flow flattening.

BDHunter correctly detects all behavior dispatchers that contain the inserted triggering conditions in samples with no obfuscations. It also successfully finds all of the behavior dispatchers in the 20 samples in the instruction substitution obfuscation. This is due to the utilization of an abstract semantic representation of the assembly instruction from lifting the original binary to IR. For the bogus control flow obfuscation, BDHunter correctly identifies the behavior dispatchers in 17 out of the 20 samples. Although this obfuscation method can add bogus basic blocks between conditional branch operations in behavior dispatchers, the path merging with *CommVar* in BDHunter can still alleviate the obfuscation to correctly detect behavior dispatchers. BDHunter fails on the 3 remaining samples due to not identifying *CommVar* in the disassembled binary. Last, for the control-flow flattening obfuscation, BDHunter detects the behavior dispatchers in 7 out of the 20 samples. This obfuscation flattens triggering mechanisms instead of forming sequential conditional branch operations with *CommVar*. Although this approach violates the dispatcher pattern that BDHunter depends on, BDHunter does detect *partial* triggering conditions in the identified behavior dispatchers in the 13 failed samples. In the worst case scenario, a malware analysis tool can still utilize the partial conditions in the behavior dispatchers to identify the corresponding trigger-based malicious behaviors. Furthermore, code normalization (composed as loop unrolling, constant propagation, and dead code elimination) can be applied to improve the resilience to the control-flow flattening further.

Real-world malware can utilize more advanced obfuscation techniques which can bypass the current version of BDHunter. However, these results empirically demonstrate that BDHunter is resilient against two of the three obfuscation techniques, and provides partial solutions against the third (control-flow flattening) obfuscation. We discuss limitations of BDHunter in §6.

5.4 Triggering Malicious Behaviors

To understand how behavior dispatchers can help reveal trigger-based malicious behaviors, we use dispatchers to guide concolic execution. Our tool is not designed to replace prior work [11, 12, 31], with regards to triggering malicious behaviors for Windows PE malware. Unfortunately, the systems and datasets from these three prior works are not publicly available. Instead, we create a baseline using angr[36] and unguided concolic execution, and compared them to the results of BDHunter-guided concolic execution.

Experiment Setup. First, we use angr, an open-source state-of-the-art binary analysis platform, as a tool for unguided symbolic execution with default exploration to observe how existing methods can explore a malware's state space. Second, we run our concolic engine on these samples without guidance toward behavior dispatchers. Instead, we provide a predefined set of suspicious API calls similar to the API-call guided approach. Then, the invocation of the set of API calls during concrete execution serve as entry-points to facilitate symbolic execution to explore malware state space. Finally, we utilize the identified dispatchers (details in §A.9) to guide the concolic execution on the same set of malware samples. In this experiment, we record the executed basic blocks from each engine and use a unified environment in order to not skew the outputs of each engine (due to different environments).

For this experiment, we utilize the 202 samples (from Table 2) of which BDHunter identified C2-related dispatchers, since C2 activities are highly related to trigger-based behaviors. Within the 202 samples, we selected 75 samples from ❶ the two largest malware families (Table 1), which are `kolabc` and `upatre` (excluding `virut` because most of the virut samples do not have C2 dispatchers in our dataset) and ❷ randomly selected seven families from the remaining families. Table 3 shows the selected families, their samples and the results of unguided symbolic and concolic execution, and BDHunter-guided concolic execution. We evaluate the three engines on the 75 samples and manually verify that the executed basic blocks contain the C2 dispatcher and trigger-based behaviors.

5.4.1 Limitations of Unguided Symbolic Execution. Unguided symbolic execution (angr) executes the samples from the original entry point for at most a 12 hours (a timeout). The reasons we set a timeout include, but are not limited to, slow emulation [5], path explosion, high memory overhead, lack of modeled API calls (leading to incorrect path formula), and the existence of complex unbounded loops (leading to complex and time-consuming path constraints). We noticed that angr suffered from the above limitations in many samples, such as consuming more than 150 GB of memory on many samples (e.g, `rbot` family). Upon hitting the timeout and a limitation on memory usage (150 GB), we save all the explored paths and basic blocks. These limitations make applying unguided symbolic execution on malware analysis difficult in practice and scalability.

The left column of Table 3 shows that angr can only reach the behavior dispatcher of the `upatre` family because the dispatcher are located in the entry function. For `shyape` and `scar`, angr cannot find complete paths to the dispatcher due to infeasible path constraints, so the path exploration is completed in less than 20 seconds. Additionally, `tedroo` contains various unmodeled API calls and encoding operations. All of these issues lead to incomplete and incorrect path formulae, which causes infeasible paths in a symbolic environment.

5.4.2 Limitations of Unguided Concolic Execution. For unguided concolic execution, similar to API-call guided approaches, we provide a predefined set of API calls and model the API calls in angr as many as we can for exploration for, at most, 15 minutes. Upon hitting the predefined API calls, the engine switches to the symbolic engine to explore in malware. However, we still noticed that it suffered the following limitations from many malware samples: ❶ the concolic engine still suffers from defining the correct set of API calls and modeling them in the symbolic engine. The engine requires human effort, which repeatedly includes additional static reverse engineering, control-flow scrutiny and dynamic traces to collect the set of essential API calls that associated with trigger-based behaviors. This effort is a tedious and time-consuming job for a human analyst. Moreover, any unmodeled API calls in symbolic engine leads to incorrect path formulae and state explosion. ❷ although an analyst can define the set of the predefined API calls and model them in the engine, the inherited limitations of symbolic execution (e.g., state/path explosion and high memory overhead from §5.4.1) are still issues before the execution reaches to the behavior dispatcher. This is because there can still be complicated checks (e.g, complex unbounded loops, encoding, and encryption) along the paths between the invoked API calls and malicious actions.

The middle column of Table 3 shows that the unguided concolic execution cannot successfully trigger malicious behaviors for `rbot`, `tedroo`, `dexter` and `kbot` (4 out of 9) due to ❷, even if a human effort is added. For example, `dexter` contains complicated checks (e.g., encoding and encryption) for the C2 commands, which cause state explosions. For `kolabc`, `shyape` and `scar` (3 out of 9), the unguided engine requires the human effort (due to ❶) to trigger malicious behaviors. The other 3 families activate all of the triggered malicious behaviors without additional human effort.

5.4.3 BDHunter-guided Concolic Execution. To assess the effectiveness of BDHunter and showcase the benefits of knowing behavior dispatcher locations, we leverage the dispatchers (details in

Family	# Samples	Unguided Symbolic Execution		Unguided Concolic Execution		BDHunter-guided Concolic Execution	
		Time	Malicious Behavior(s)	Time	Malicious Behavior(s)	Time	Malicious Behavior(s)
kolabc	49	<5 secs	No hidden behavior(s)	15 mins	Extfiltration over Web Service (HE*)	15 mins	Extfiltration over Web Service
upatre	11	>12 hrs	Download and Execute new malware	15 mins	Download and Execute new malware	15 mins	Download and Execute new malware
shyape	5	<20 secs	No hidden behavior(s)	15 mins	Download and Execute new malware (HE*)	15 mins	Download and Execute new malware
scar	3	<20 secs	No hidden behavior(s)	15 mins	Download and Execute new malware (HE*)	15 mins	Download and Execute new malware
rbot	2	<45 mins	No hidden behavior(s)	15 mins	No hidden behavior(s)	15 mins	Data Collection and Extfiltration over C2 channel
mydoom	2	>12 hrs	No hidden behavior(s)	15 mins	Drop and Execute new malware	15 mins	Drop and Execute new malware
tedroo	1	>12 hrs	No hidden behavior(s)	15 mins	No hidden behavior(s)	15 mins	Download and Execute new malware Copy and Spread the dropped file(s)
dexter	1	>12 hrs	No hidden behavior(s)	15 mins	No hidden behavior(s)	15 mins	Multi-stage C2 Channels Download and Execute new malware Data Destruction and Service Stop
kbot	1	>12 hrs	No hidden behavior(s)	15 mins	No hidden behavior(s)	15 mins	Download and Execute new malware Service Stop

Table 3: Times and triggered malicious actions by unguided symbolic & concolic execution, and BDHunter-guided concolic execution. The table shows that BDHunter-guided concolic execution reveals 13.0× and 2.6× more trigger-based malicious behaviors, compared to unguided symbolic and concolic execution. The malicious behaviors in this table are triggered after hitting the behavior dispatcher from each engine. HE* means that the malicious behavior(s) requires human effort to be activated in the execution engine. This effort includes additional static reverse engineering, control-flow scrutiny, and dynamic tracing to collect the set of essential API calls that are related to the malicious behavior(s), but are hidden by complicated checks.

§A.9) for concolic execution (described in §4) and allow the execution to explore for, at most, 15 minutes. This evaluation demonstrates that BDHunter-guided concolic execution can overcome the limitations of unguided symbolic and concolic execution and activate the trigger-based behaviors. Unlike unguided concolic execution, the only human effort required is to analyze the top candidate dispatchers and feed the basic block address having the dispatchers to the concolic execution. As we described in §5.1, the dispatchers are identified within top 20 candidates from 77.4% of our dataset.

The right column of Table 3 shows that BDHunter-guided concolic execution unfolds 13.0× and 2.6× more trigger-based malicious behaviors, compared to unguided symbolic and concolic execution. BDHunter-guided concolic execution successfully activates the trigger-based behaviors including the families (e.g., `rbot`, `tedroo`, `dexter` and `kbot`) that the unguided approaches failed to activate to malicious behaviors. BDHunter-guided engine is guided towards the the identified dispatcher, identify triggering conditions with a trigger value, and fulfill the conditions to activate the trigger-based malicious actions compared to the baseline. Next, we provide two case studies from different families to highlight the effectiveness of BDHunter-guided concolic execution.

Case Study 1: Dexter. This case study covers `dexter` from §A.9. This malware sample is a bot that interacts with a C2 server. The comparison results for this malware are shown in the 8th row in Table 3. With BDHunter-guided concolic execution, we explore three C2 command-related behaviors which are unseen during the two unguided executions. For example, one behavior is *Multi-Stage C2 Channels*, which represents multiple stages for C2 that are employed under different conditions. On the other hand, both unguided executions fail to reach the behavior dispatchers due to the path explosion issue. Although the unguided concolic engine is supported by human effort to set the necessary API calls, it fails to expose the trigger-based behaviors. Additionally, the sample contains the complicated checks (e.g., encoding and encryption) and the loops (inside of the checks) after receiving C2 commands, which cause state explosion issues.

Case Study 2: kolabc. This case study covers `kolabc` (details in §A.9). This malware sample mainly performs *Data exfiltration*, which attempts to steal data from the infected endpoint. The comparison results on this malware are shown in the first row of Table 3. It fingerprints the host information (e.g., locale and IP) and waits for a command from the C2 server. BDHunter-guided concolic execution reaches the dispatcher and feeds in the trigger inputs. Then, it sends the collected files through socket API calls (e.g., `connect` and `send`). On the other hand, the unguided symbolic engine fails to trigger behaviors because of its incomplete features to simulate a heap operation for the PE binary. However, for the unguided concolic engine, after human effort is added to feed the essential set of API calls, the engine activates the behaviors.

6 Limitations and Discussion

Evading Dispatcher Pattern. Adversaries aware of the existence of BDHunter can deliberately obfuscate dispatching patterns to evade it. Although we demonstrated the robustness of BDHunter in §5.3, an adversary may obfuscate the patterns by control-flow obfuscation or by forcing each conditional branch operation to use a different variable or moving the operations into multiple functions. This would affect the efficiency and the accuracy of BDHunter. However, this limitation is similar to other pattern-based approaches that rely on static analysis [4, 15, 24].

To overcome this problem, we could apply code normalization in IR, including loop unrolling, constant propagation, and dead code elimination. These transformations can eliminate the additional variable and conditional branches introduced from the control-flow obfuscation. Besides, the normalization of dynamic execution with taint for value flow analysis is also applicable, if an adversary attempts to use a different variable in each conditional branch operation or move the operations in multiple functions. By employing value numbering in the dynamic execution, the value flow analysis can identify the variables and memory address that carry the same value. This will also help analyze the function pointers that are passed through value flows and benefit call site identification.

Like an arms race, malware authors will eventually find ways to bypass our methods. However, they need to carefully adapt more targeted obfuscation techniques because they are required to not only manipulate their dispatcher patterns, but also ensure that the functionalities of the malware are preserved. This increases the difficulty of creating new malware to bypass our system.

Additionally, adversaries can conceal triggering conditions and trigger-based behaviors by encryption-based obfuscation [34]. This can provide a strong guarantee to hide behaviors by encrypting trigger-based code with a key and using the combinations of inputs as the decryption key to trigger behaviors. However, this is both a limitation of BDHunter *and* other approaches that activate trigger-based behaviors by fulfilling or forcing triggering conditions [30]. However, these malware are rare in practice [25]. Another rare case that BDHunter cannot handle is malware that uses a customize virtual instruction set architecture (ISA) or an interpreted meta language, which makes it more difficult to accurately identify dispatcher patterns.

Identifying Dispatcher Pattern. To accurately identify dispatcher patterns, BDHunter assumes an accurate disassembly to extract instructions, basic blocks and functions. However, there are well-known limitations [1, 2] of accurately disassembling. There could also be multiple ways of implementing a conditional branch operation for comparison, which is a key component of behavior dispatcher. BDHunter might miss some instructions that involve comparison due to the complexity of x86 instruction set (e.g., `CMPXCHG` is branchless instruction but involves comparison).

To overcome this problem, BDHunter relies on BinaryNinja, an industry-standard disassembler, and its IR (BNIL) which provides a more abstract semantic representation of x86 assembly [39]. Thus, BDHunter utilizes these higher-level semantics to identify related instructions for a conditional branch operation and a common value at the best-effort. Moreover, additional instructions and patterns can be added as BDHunter is configurable.

Accuracy of BDHUNTER. The three thresholds (in the pattern-based method) and the set of suspicious APIs (in the weight-based method) are used by BDHunter to efficiently detect dispatcher patterns. Although these configurations performed well in the 302 real-world malware samples, it is possible that other malware samples could cause more FPs and FNs. Also, an adversary may target these configurations to subvert BDHunter's accuracy.

BDHunter can be configured to adjust its accuracy as more and different malware samples are encountered. Further, additional patterns and dynamic information (e.g., dynamic tainting a common value) can be added to enhance the accuracy of BDHunter. With the additional methods above, BDHunter can achieve higher accuracy of identification across more malware samples.

7 Related work

Evasion Detection. To evade or mislead analysis, malware often contain multiple evasive techniques such as virtualization and sandbox system detection logic [26]. Trigger-based malicious behaviors tend to be located after evasive techniques. Thus, detecting the techniques is an important problem to perform malware analysis. Existing approaches [23, 24] attempt to detect evasive behavior by applying differential analysis to find dynamic behavior deviation in different environments and artifact-based signature scanning. These approaches are effective in detecting known evasive techniques. Unlike these approaches, BDHunter specifically focuses on identifying behavior dispatchers, which indicate locations that require trigger values to activate malicious behaviors.

Cryptographic-encoding Detection. Malware often uses cryptographic and encoding functions for various reasons, such as evading analysis and performing malicious activities like C2 communication. Detecting such functions is important because malware analysis based on symbolic execution are limited by these functions. Previous work [12] detects the cryptographic functions for solving constraints. However, identifying cryptographic and encoding functions still remains a challenge (e.g, [12] is required to identify the inverse of encryption functions, which cannot be done for common encryption algorithms like RSA). Unlike previous work, BDHunter does not try to detect cyrptographic-encoding functions, but rather focuses on identifying dispatchers to reach the malicious behaviors by skipping them as much as possible.

Multi-path Exploration and Forced Execution. Another popular malware analysis technique is to explore multiple execution paths via concrete executions [11, 28, 31]. These technique can effectively achieve high code coverage in analyzing malware. However, high code coverage does not necessarily achieve reaching the core logic to trigger its behaviors. Also, it is well-known that performing high code coverage is computationally expensive [15]. For example, [28, 31] require too many resources and time to cover many execution paths and are difficult to deploy at scale in practice. [11] requires manual effort to pinpoint the symbolic source from predefined API calls. Unlike prior works, BDHunter adopts a target-specific approach by identifying dispatchers and using them to trigger malicious behaviors in practice. The findings of BDHunter are designed to give insights about malware to focus and guide its execution to reveal more behaviors. BDHunter feeds the identified dispatchers to the executing engine to guide its path exploration, so that the engine will not blindly explore the malicious sample, and we will not assume that the execution will luckily hit the core logic of the malware.

Pattern-based Behavior Detection. Pattern-based detection approaches [4, 15, 35] have been widely applied to malware analysis and vulnerability assessment. [15] constructs behavior models based on observed malicious behaviors while dynamically executing, and uses those models to statically identify the capabilities of malware via code structure comparison. However, this approach relies on the coverage of the model set, which can be limited for unknown malware capabilities. [35] performs vulnerability assessment for firmware in embedded devices. It uses symbolic execution on statically sliced program binaries to produce constraints and performs constrains solving via manually provided privileged program points. While this approach improved the automation of vulnerability detection for firmware, it still cannot handle complex checks and requires manual annotations. By comparison, our pattern-based method does not rely on prior knowledge about the malware, and focuses on dispatcher patterns for trigger-based malware.

8 Conclusion

In this paper, we propose BDHunter, a system that automatically identifies behavior dispatchers, a common component that distributes a malware's execution to different actions in trigger-based samples. In our evaluation, we demonstrate that BDHunter is effective and accurate in identifying behavior dispatchers. Furthermore, the identified behavior dispatchers are useful for assisting concolic execution engines in revealing trigger-based malicious actions.

9 Acknowledgment

We thank our shepherd, Kehuan Zhang, and the anonymous reviewers for their helpful and informative feedback. This material was supported in part by the Office of Naval Research (ONR) under grants N00014-17-1-2895, N00014-15-1-2162, and N00014-18-1-2662, and the Defense Advanced Research Projects Agency (DARPA) under contract HR00112090031. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of ONR or DARPA.

References

[1] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. 2016. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th USENIX Security Symposium USENIX Security 16*. 583–600.

[2] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 177–189.

[3] AV-TEST - The Independent IT-Security Institute. 2020. Statistic Malware samples in 2020. https://www.av-test.org/en/statistics/malware/.

[4] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically-directed Dynamic Automated Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 12–22.

[5] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Journal ACM Computing Surveys (CSUR) Surveys Homepage archive* 51, 50 (2018).

[6] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 189–200.

[7] Sebastian Banescu, Martín Ochoa, and Alexander Pretschner. 2015. A framework for measuring software obfuscation resilience against automated attacks. In *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE, 45–51.

[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.

[9] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. 2012. Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies. In *Black Hat USA Briefings (Black Hat USA)*. Las Vegas, NV.

[10] Derek L. Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation. Cambridge, MA, USA. AAI0807735.

[11] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. 2008. Automatically Identifying Trigger-based Behavior in Malware. In *Botnet Detection*. Springer, 65–88.

[12] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Domagoj Babi ć, and Dawn Song. 2010. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM conference on Computer and communications security*. 413–425.

[13] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. 2018. Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 395–411.

[14] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices* 46, 3, 265–278.

[15] Paolo Milani Comparetti, Guido Salvaneschi, Eggngin Kirda, Clemens Kolbitsch, Christopher Kruegel, and Stefano Zanero. 2010. Identifying Dormant Functionality in Malware Programs. In *2010 IEEE Symposium on Security and Privacy*.

[16] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*. 51–62.

[17] FalconSandbox. 2020. Hybrid-Analysis. https://www.hybrid-analysis.com/.

[18] Nicolas Falliere and Eric Chien. 2019. Zeus: King of the Bots. https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/security-response-zeus-king-of-bots-09-en.pdf.

[19] Hex-Rays. 2020. IDA F.L.I.R.T. https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml.

[20] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. 2019. Fuzzification: Anti-fuzzing techniques. In *28th USENIX Security Symposium USENIX Security 19*. 1913–1930.

[21] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM–software protection for the masses. In *2015 IEEE/ACM 1st International Workshop on Software Protection*. IEEE, 3–9.

[22] Jacob Kastrenakes. 2019. Agent Smith Malware Has Replaced Android Apps' Code on 25 Million Devices. https://www.theverge.com/2019/7/10/20688885/agent-smith-android-malware-25-million-infections.

[23] Dhilung Kirat and Giovanni Vigna. 2015. Malgene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 769–780.

[24] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. 2011. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM conference on Computer and communications security*. 285–296.

[25] Eugene Kolodenker, William Koch, Gianluca Stringhini, and Manuel Egele. 2017. Paybreak: Defense against cryptographic ransomware. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 599–611.

[26] LordNoteworty. 2020. Al-Khaser. https://github.com/LordNoteworthy/al-khaser.

[27] Masrepus vfsrfs garanews. 2020. Unpacking PE files using Unicorn Engine. https://github.com/unipacker/unipacker.

[28] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Exploring multiple execution paths for malware analysis. In *2007 IEEE Symposium on Security and Privacy*. IEEE, 231–245.

[29] Yuval Nativ. 2021. theZoo. https://thezoo.morirt.com/.

[30] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. 2019. How to kill symbolic deobfuscation for free (or: unleashing the potential of path-oriented protections). In *Proceedings of the 35th Annual Computer Security Applications Conference*. 177–189.

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

[32] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*. IEEE, 317–331.

[33] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. 2016. Avclass: A tool for massive malware labeling. In *International symposium on research in attacks, intrusions, and defenses*. Springer, 230–253.

[34] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. 2008. Impeding Malware Analysis Using Conditional Code Obfuscation.. In *NDSS*. Citeseer.

[35] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware.

[36] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 138–157.

[37] The Linux Foundation. 2020. gRPC. https://grpc.io/about/.

[38] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. 2015. SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 659–673.

[39] Vector35. 2020. BinaryNinja Intermediate Language. https://docs.binary.ninja/dev/bnil-llil.html.

[40] VirusTotal. 2021. VirusTotal. https://virustotal.com/.

[41] Wine. 2020. Wine API to Forward Windows API. https://source.winehq.org/WineAPI/.

[42] Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2017. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 921–937.

[43] Fratantonio Yanick, Bianchi Antonio, Robertson William, Kirda Engin, Kruegel Christopher, and Vigna Giovanni. 2016. TriggerScope: Towards detecting logic bombs in android applications. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 377–396.

[44] Michal Zalewski. 2019. American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl.

Appendix A

A.1 Categories of behavior dispatchers.

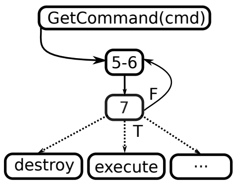
Behavior dispatcher	Features
C2 command	Trigger an activity that leads to reconnaissance, initial compromise, control, exfiltration, and et al
File Systems	Create/read/modify/delete file, File size, Paths
Process	Create/read a process, Loading Library, Process memory
Network	Requests and responses, DNS, Connections
Registry	Create/read/delete keys
Time	System Time check, Time sleep
Environment	OS version, AV software, VM awareness

Table 4: Although BDHunter identifies general dispatching behaviors, the above is a table of different categories that make up the behaviors it found.

A.2 Behavior dispatcher with Zeus's code.

```
1 struct {short Id; CMDProc proc;}
2 cmds[MAX] ={{CMD_DESTROY, destroy},
3            {CMD_EXECUTE, execute},...};
4 if (GetCommand(cmd) != 0) return -1;
5 for (int ci = 0; ci < MAX; ci++) {
6   char *name = map(cmds[ci].Id);
7   if(_lstrcmpiW(name, cmd) == 0)
8     cmds[ci].proc(cmd);
9 }
```

(a) Source code



(b) Control-flow graph

Figure 4: Malware Zeus uses a loop for command comparison and dispatch.

A.3 Malware samples and their families used in training.

Label	(#3) zbot, (#3) dexter, (#2) blakken, (#2) dinwod, (#2) emotet, (#2) gamarue, (#2) mydoom, (#2) rbot, (#2) regin, (#2) sinowal, (#2) vobfus, (#1) alinaos, (#1) blackenergy, (#1) carbank, (#1) carberp, (#1) cometer, (#1) duqu, (#1) fakealert, (#1) fonten, (#1) fuerboos, (#1) gravityrat, (#1) hupigon, (#1) installerex, (#1) khalesi, (#1) kovter, (#1) njrat, (#1) poscardstealer, (#1) sekur, (#1) sofacy, (#1) stuxnet, (#1) tedroo, (#1) zemra, (#1) zeroaccess, (#1) zonidel, (#1) zusy (#2) singleton,
Total	38

A.4 Malware samples and their families used in evaluation.

Label	(#49) kolabc, (#34) virut, (#11) upatre, (#7) zbot, (#6) chir, (#6) crytex, (#6) scar, (#6) zegost, (#5) Hematite, (#5) sality, (#5) shyape, (#4) bladabindi, (#4) gamarue, (#4) Rodecap, (#4) tinba, (#3) expiro, (#3) Qakbot, (#3) ursnif, (#3) ursu, (#3) wabot, (#3) wapomi, (#3) zusy, (#2) agen, (#2) autoit, (#2) badur, (#2) blackhole, (#2) cosmu, (#2) delf, (#2) hworld, (#2) ibryte, (#2) ircbot, (#2) nitol, (#2) pioneer, (#2) qbot, (#2) qqpass, (#2) Ramnit, (#2) rbot, (#2) regrun, (#2) tempedreve, (#2) trickbot, (#2) unruy, (#2) vanbot, (#2) vobfus, (#2) wlksm, (#1) acidhead, (#1) acidsena, (#1) adonai, (#1) agobot, (#1) aladin, (#1) aleph, (#1) alicia, (#1) almaster, (#1) alphabot, (#1) amanda, (#1) amitis, (#1) angelfire, (#1) anibot, (#1) antes, (#1) antilam, (#1) apbost, (#1) aslyum, (#1) assasin, (#1) backend, (#1) blackengery, (#1) brabot, (#1) Brambul, (#1) breplibot, (#1) chinabomb, (#1) cmjspy, (#1) dancerbot, (#1) danton, (#1) darkbot, (#1) daws, (#1) dealply, (#1) delphi, (#1) destructbot, (#1) dexter, (#1) diewar, (#1) downloadersponsor, (#1) duke, (#1) eggdrop, (#1) emotet, (#1) feardoor, (#1) funlove, (#1) gandcrypt, (#1) gator, (#1) harebot, (#1) horst, (#1) hosts, (#1) icedid, (#1) icmp, (#1) installmonster, (#1) jaiko, (#1) joiner, (#1) joke, (#1) kbot, (#1) lamebot, (#1) latinus, (#1) leetbot, (#1) nanocore, (#1) offend, (#1) parite, (#1) pykspa, (#1) redfox, (#1) sadenav, (#1) seimon, (#1) simbot, (#1) socelars, (#1) speedingupmypc, (#1) startsurf, (#1) tiggre, (#1) ultimaterat, (#1) virlock, (#1) vnuke, (#1) vtflooder , (#1) zapchast, (#9) Singleton
Total	127

A.5 Selected suspicious API calls for the weight-based identification.

Resource	API calls
File	CreateFile, WriteFile, ReadFile, DeleteFile
Process	LoadLibraryW, GetProcAddress, ShellExecuteW, ShellExecuteA, ExitProcess, GetProcessHeap, OpenProcess, CreateWindowExA, CreateProcessA, Process32First
Network	InternetReadFile, InternetOpen, InternetConnect, InternetSetOption, InternetGetConnectedState, InternetQueryOption, InternetOpenUrl, InternetConnect, InternetQueryOption, InternetCreateUrl, InternetWriteFile, InternetReadFileEx, InternetSetCookie, InternetSetOptionEx, InternetCombineUrl, HttpSendRequest, HttpQueryInfoW, HttpOpenRequestW, WinHttpSendRequest, Ordinal_WSOCK32_5, Ordinal_WSOCK32_107, Ordinal_WSOCK32_20, InternetCanonicalizeUrl, MSWSOCK, wsock32, WinHttpGetDefaultProxyConfiguration, closesocket, ioctlsocket, gethostname, getsockname, SendInput, htons, WSARecv, WSASend, WSASocketA, WSASocketW, WSARecvFrom, recvfrom, recv, sendto, send, WININET, WINHTTP, DnsFree, DnsQuery_A, DnsQuery_UTF8,
Registry	RegEnumKeyEx, RegSaveKey, RegRestoreKey, RegLoadKey, RegSetValue
Other	lstrcmpiW, lstrlenW, lstrcmpW, lstrcpyA, strncpy, strncmp, strcmp, strstr, strnicmp, wcscmp, LoadStringW, LCMapStringW, LCMapStringA, CompareStringW, CompareStringA, GetEnvironmentStringsW, SetEnvironmentVariableW, FreeEnvironmentStringsW, timeSetEvent, GetCPInfoExW, CryptAcquireContextA

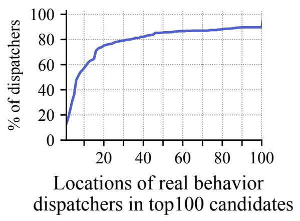
A.6 Algorithm of weight-based dispatcher identification.

Algorithm 2: Weight-based Dispatcher Identifier

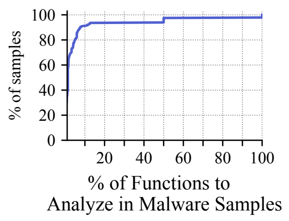
```
Input  : CFGs :- control-flow graphs of the whole program
         Map :- a map from suspicious APIs to their weights
Output : S :- the set of candidate behavior dispatcher

Procedure WeightBasedIdentifier(CFGs, Map)
    while update do                              // propagate weights to callers
        update = 0;
        foreach CFG ∈ CFGs do                    // check each function
            hasUnknownn = 0; weight = 0;
            foreach callInst ∈ CFG do            // check each callee
                target = callInst.getCallTarget();
                if target ∉ Map then
                    hasUnknown = 1; break;
                weight += Map[target];
            if hasUnknown then continue;
            Map[CFG] = weight;
            update = 1;
    foreach CFG ∈ CFGs do                        // diff callers and callees
        foreach F ∈ CFG.getCallees() do
            CFG.diff_callees += Map[CFG] / Map[F];
        CFG.diff_callees /= CFG.getCallees().length();
        foreach F ∈ CFG.getCallers() do
            CFG.diff_callers += Map[F] / Map[CFG];
        CFG.diff_callers /= CFG.getCallers().length();
        CFG.weight_diff = CFG.diff_callees / CFG.diff_callers;
        if CFG.weight_diff > W then S.add(CFG);
    return S.sort();
```

A.7 Distribution of identified dispatchers and functions that contain dispatchers.



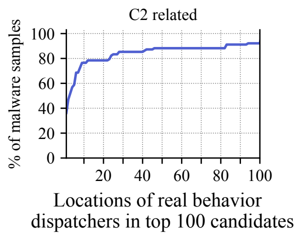
(a) CDF of behavior dispatchers



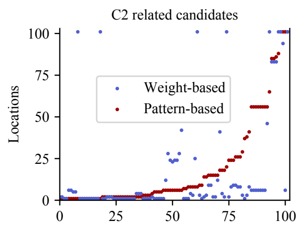
(b) CDF of functions

Figure 5: (a) shows we identify a large majority of dispatchers within the top 20 dispatchers. (b) shows we identify a majority of samples' dispatching functions within the top 10% of all functions.

A.8 Comparison between pattern-based and weight-based methods in C2-dispatchers.



(a) CDF of samples



(b) Comparison

Figure 6: (a) shows the CDF of samples with C2-based dispatchers among the top 100 candidates, reported by the weight-based method. (b) shows the comparison pattern-based and weight-based methods on detecting C2-based dispatchers in 102 samples.

A.9 Selected samples per family and their behavior dispatchers that used in evaluation.

SHA256	Family	Simplified Behavior dispachers			Year (first submission in Virustotal)
		Type	Function	Basic block chains [Start, End]	
001017c87a1a4c9aca8b55fe265babb5c3a97463bab33588a0a83925d8fc7d95	upatre	C2-related	0x401000	[0x401024, 0x4011cc]	2018
02186ad0ea9680bef3e6ea28d08c085332cba41c8f7392af7134cd553ca3d289	ibryte	C2-related	0x402032	[0x402252, 0x408734]	2018
044bb96c58ca05fa63a5292a909604604f499f6f0d334444e63d62b04e3a97dd	kolabc	C2-related	0x405338	[0x4053b6, 0x40570f]	2018
052ffac86ad7db839562e1c6b578bd732e363986e91ec67e67d2eb41213feb3c	gator	C2-related	0x409a20	[0x409b4d, 0x409c99]	2018
054027e32a23362e2b26d6d3081e4a4c3b400bca6ada3ea76e940099c6b14409	expiro	C2-related	0x10022bc	[0x1002310, 0x100255d]	2018
10cea7d2289c53e4d432762611906783afa9735e493471d4f9f3eb5a63ba0d0a	wapomi	C2-related	0x4095f4	[0x409638, 0x409e82]	2018
115c24b2e7ac5ebefcdf064c7813d988315cc4c489963962d3880474e9bcdbaf	autoit	C2-related	0x40a000	[0x40ac03, 0x40b6d1]	2018
121072e3e6bb51ce21911e17a2dc7555e5d71ff1de15262cfcf99bc1aac64c99	virut	C2-related	0x10050ce	[0x1005124, 0x100598e]	2018
14e02de1cd46b1b0198f73f07262a3847d0ce1e7871461202252b95b73bc5aba	sality	C2-related	0x30008714	[0x30008714, 0x30008957]	2018
1c4bb5d1896d1cbf495f15dfe14dee80decde7e01027ddd168801deffebdf3c0	chir	C2-related	0x1001f41	[0x1001f67, 0x10021eb]	2018
1cc1bd952ad29e44a1a249d9d9f9492048acfdc77fc220fb8296f28b47749a19	host	C2-related	0x40e800	[0x40e810, 0x40f1a6]	2018
1e2a3f5a1f727b2b6072adefb992c3da51dcf2cf85d1491713640a8d95ddb063	downloadersponsor	C2-related	0x46638d	[0x46640b, 0x466806]	2018
22cb952cce7ffdc74b99c1c961c725a31c9c8b1032136cf279448078f2762ccc	cosmu	C2-related	0x40eaa4	[0x40eb31, 0x40edfc]	2013
2356d6bac437beef7b854336594e0d262a7a91c02941b6be27d993fd6991b188	wabot	C2-related	0x405cdc	[0x405ce5, 0x406180]	2018
2df5bbe0e055e2af7d32e3b71ea80b70f844a917229a6b7f9668eca31c3d813e	zbot	C2-related	0x410650	[0x410761, 0x4107d5]	2012
2dfa173806d9a0da60bb4a2b0fa4ff0979543d80a188e7c81d4590ee26ac2a39	lamebot	C2-related	0x402661	[0x402707, 0x4028fc]	2015
2fc9b799fe335e563b9d838d163c6c7d045823c4366b0ff411969a5b265aa06d	hworld	C2-related	0x1004ae8	[0x1004b35, 0x1004bef]	2018
318f149602a8a8f9049d851f5d12f41e3b84c6ae48159cfd2a27b62f843e1398	speedingupmypc	C2-related	0x40e77c	[0x40e7a4, 0x40e865]	2018
3fa55852b0974ab0c4ef3db11963da466835787009d769c089e299cf9dcb322d	unruy	C2-related	0x40f45e	[0x40f4a7, 0x40fa39]	2018
40d3fe54bde382254fc2f562af2fc597279e4ebe9d717f0f828b9a414612d730	bladabindi	C2-related	0x404c50	[0x404e75, 0x404f5a]	2018
43933b0112dcf49c748246b2b4abe066ff067d962d590c4c5b82b03af4b0425e	shyape	C2-related	0x402900	[0x402923, 0x402a31]	2018
457b4058dab6f5a55666d52f3359e609c96a2c7a0f9f20e8b2163b8cbb51b990	mydoom	C2-related	0x804d32	[0x804d6a, 0x804efa]	2019
46eeb9690c04a406c0bbb1aa3dbd9e04a5a61bedce2ecc456ab3be3ed3182f2b	Hematite	C2-related	0x1004ae8	[0x1004b35, 0x1004bef]	2018
49cce631a6c2135f7a4ceefac9260bdd65ae903cdde25d2747fe52bb5b4b53e3	nitol	C2-related	0x401be0	[0x401cb2, 0x402160]	2018
4a8a92d9bd345ddee3134702db9e0fe573847c20ea0ba335c406b5cba46335df	badur	C2-related	0x4010a0	[0x4010a0, 0x401203]	2018
4c328f22bbbe056c489b5dee595d657180870bddbffef852d2bb672926f5cac4	offend	C2-related	0x40e010	[0x40e080, 0x40e510]	2018
4eabb1adc035f035e010c0d0d259c683e18193f509946652ed8aa7c5d92b6a92	dexter	C2-related	0x405ca0	[0x405cc1, 0x405e9c]	2013
4f0e0f8f17a74f12457e9353ee87324feb503b1bc3e7a025b8d2bca918aff939	Qakbot	C2-related	0x4019f7	[0x40b711, 0x40c293]	2018
5ee455952368a1a80cb1724387636f74af92ab92e1334f997aabd7e7846006de	adonai	C2-related	0x48c1dc	[0x48c20a, 0x48deec]	2013
5fed3475b4498a1d130996cbe268273a15b46a1a3060c56cb8ecda4b3e80f97a	Ramnit	C2-related	0x413d20	[0x4146a4, 0x414b06]	2018
608cb84a2277d4b7222af0293a1b682142c913d63097331ecae73e2fce7ea155	Rodecap	C2-related	0x4130c0	[0x41330d, 0x41374b]	2018
6a340dbff8c4d6f557ea5882227379ac2ea7a080d1da4204ef3a3b24fa6af877	ursu	C2-related	0x4019f7	[0x401cc2, 0x401d70]	2018
6a677e19dd0adb0902be09094cb2e64a1e1e422b25b8d2d24adac95640dede1a	startsurf	C2-related	0x4914c0	[0x4914f3, 0x491cb7]	2018
6bb31105cd051824bdab997ba4af0aeac9aff8deebdd637e480f3b39e3905802	tinba	C2-related	0x403520	[0x4037ce, 0x40447a]	2018
6be9c723d766a4478a74f7d3ba722dfd37ac120e1267d3da3fa88ff841a936e0	crytex	C2-related	0x100133f	[0x1001362, 0x10013db]	2018
6e50e0137ef2e57fa16d2750a1eff8f0b8552a29f48bad8e41196e39c71f458c	zusy	C2-related	0x40ddfd	[0x40e407, 0x40e9aa]	2018
72e5f89846cb2573b07a32e1c106dfbdbd56cd1a593836577f062401bcfac9fc	scar	C2-related	0x402900	[0x402940, 0x402a6f]	2018
8bb8feb3d5fb92c7584f1ccb94459307376dae239ce0beb459427efae0db905e	tedroo	C2-related	0x405b30	[0x405c26, 0x406319]	2019
8c0736746c5c70cafd2510318ec8d1a34192ef877128dbd9d4ba6db0e3c80d46	ircbot	C2-related	0x407a90	[0x407d61, 0x402377]	2018
8c86f78dc42fe98cf2aa0e412515e35ed4178f2d8fc066d2e7344dd91406675f	gandcrypt	C2-related	0x4011f8	[0x401231, 0x4014cb]	2018
9603098860e28858233ad7badfb3c76773f68a734bd12ceee11b7bb9e2f3b7be	vtflooder	C2-related	0x4010a0	[0x4010dc, 0x401203]	2018
a0a0c779dadcdd3328585316aff9b838f64b7afa070f4a63fe85bca747cb1d88	nanocore	C2-related	0x41a7bb	[0x41a854, 0x41b384]	2018
aebeb5363ad2aea39960fa2422241204a557259fcad8df60ee46eb902f53061e	alicia	C2-related	0x4743a8	[0x474401, 0x4746a6]	2013
b11c5fa939db2157c36c0a3a92966388dc81573a236da224753b8613959c7dcc	pykspa	C2-related	0x40fa36	[0x40fa63, 0x40fd1a]	2018
b5610d256505e8f590318cc891ca6277452f421b9d4b5a4551508ebddeef3d95	agen	C2-related	0x40164f	[0x402965, 0x4037e6]	2018
be40534b3738098b0ad386389c93ac9d3a7f89c592faa194a6a802d7f1b28880	zegost	C2-related	0x41eff0	[0x41f0cb, 0x41f248]	2018
c0c6be94ec07488ef2bd69894ba6c4955fbf5979599c47a1a68a76d719e3eac0	rbot	C2-related	0x40c580	[0x40c6e5, 0x40ca7b]	2018
c168184d91abc235db97c95163fabd64b8273069e99571fbc3fc0a6f2726f7df	kbot	C2-related	0x15112a00	[0x15112a00, 0x15112d09]	2019
c2470aa1143e956fe2d358a76f1c1bd2159bf9c98741a8b2b2e94516cdd852bb	agobot	C2-related	0x40138e	[0x401437, 0x4028f2]	2012
f689947fb17d061d8e49751efc84129a7ca0906a2adbd68e3909448f58201d7e	pioneer	C2-related	0x40138e	[0x45a4ae, 0x45a6e2]	2018