Define-Use Guided Path Exploration for Better Forced Execution

Dongnan He
Renmin University of China
Beijing, China
hedongnan@ruc.edu.cn

Dongchen Xie
Renmin University of China
Beijing, China
dongchenx@ruc.edu.cn

Yujie Wang
Renmin University of China
Beijing, China
wyujie@ruc.edu.cn

Wei You*
Renmin University of China
Beijing, China
youwei@ruc.edu.cn

Bin Liang
Renmin University of China
Beijing, China
liangb@ruc.edu.cn

Jianjun Huang
Renmin University of China
Beijing, China
hjj@ruc.edu.cn

Wenchang Shi
Renmin University of China
Beijing, China
wenchang@ruc.edu.cn

Zhuo Zhang
Purdue University
West Lafayette, USA
zhan3299@purdue.edu

Xiangyu Zhang
Purdue University
West Lafayette, USA
xyzhang@cs.purdue.edu

ABSTRACT

The evolution of recent malware, characterized by the escalating use of cloaking techniques, poses a significant challenge in the analysis of malware behaviors. Researchers proposed forced execution to penetrate malware's self-protection mechanisms and expose hidden behaviors, by forcefully setting certain branch outcomes. Existing studies focus on enhancing the forced executor to provide light-weight crash-free execution models. However, insufficient attention has been directed toward the path exploration strategy, an aspect equally crucial to the effectiveness. Linear search employed in state-of-the-art forced execution tools exhibits inherent limitations that lead to unnecessary path exploration and incomplete behavior exposure. In this paper, we propose a novel and practical path exploration strategy that focuses on the coverage of define-use relations in the subject binary. We develop a fuzzing approach for exploring these define-use relations in a progressive and self-supervised way. Our experimental results show that the proposed solution outperforms the existing forced execution tools in both memory dependence coverage and malware behavior exposure.

CCS CONCEPTS

• **Security and privacy** → **Malware and its mitigation**; • **Software and its engineering**;

KEYWORDS

dynamic analysis, forced execution, path exploration

*corresponding author

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3652128

ACM Reference Format:
Dongnan He, Dongchen Xie, Yujie Wang, Wei You, Bin Liang, Jianjun Huang, Wenchang Shi, Zhuo Zhang, and Xiangyu Zhang. 2024. Define-Use Guided Path Exploration for Better Forced Execution. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3652128

1 INTRODUCTION

Malware poses a persistent and significant threat to individuals, organizations, and even nations. Recent statistics show that malware is responsible for 80% of cyber-attacks across the globe [14], causing $10.5 trillion financial loss per year [6]. In order to effectively defend malware, the security community needs to perform a critical task of analyzing malware behaviors. However, it is difficult to fully disclose malware behaviors via plain execution in a sandbox [3, 5], as malware authors are increasingly focused on developing stealthy malwares that leave lighter footprints [2, 4, 8]. Specifically, a malware often requires a specific execution environment or setup, which may be absent (e.g., the command and control server is down and a critical registry key is missing). In addition, recent malware often defines very specific conditions (e.g., time-bomb and logic-bomb) to release payload. Some sophisticated samples even use cloaking techniques (e.g., packing and VM/debugger detectors) to prevent execution upon the presence of analysis environment.

To penetrate malware self-protection and expose hidden behaviors, researchers proposed a technique called *forced execution* that works by forcefully setting certain branch outcomes. Since forcing execution paths could lead to corrupted states and hence exceptions, the primary challenge lies in ensuring *crash-free execution*. X-Force [40] achieves this in a heavy-weight manner that allocates a new memory block on demand upon any invalid pointer dereference. It requires tracing individual memory and arithmetic instructions, reasoning about pointer alias relations, and repairing invalid pointers on-the-fly. PMP [52] adopts a light-weight approach that pre-allocates a large memory region and fills the region with carefully crafted random values before execution. These values are designed in such a way that dereferencing an invalid

```
1  #define REG_URL "regular-site.com"
2  #define MAL_URL "malicious-site.com"
3  #define VALUE "ScreenRibbonsDomain"
4  #define KEY "SOFTWARE\\...\\ScreenSavers\\"
5  #define DEST_PATH "C:\\...\\Roaming\\eset_update.exe"
6  #define LEN 0x104
7
8  typedef enum {
9    OP_NONE,
10   OP_DOWNLD,
11   ...
12 } Opcode;
13
14 typedef struct {
15   int op;
16   char arg[LEN],
17   url[LEN];
18 } Command;
19
20 Command cmd;
21 HKEY phk;
22 char file_path[LEN], exe_path[LEN], msg[LEN];
23 int main() {
24   exe_path = get_current_path();
25   if (strcmp(exe_path, DEST_PATH))
26     CopyFile(exe_path, DEST_PATH);
27   if (RegOpenKeyEx(..., KEY,..., &phk) == OK) {
28     if (RegQueryValueEx(phk, VALUE, &cmd.url,...) != OK) {
29       memset(cmd, 0, sizeof(cmd));
30       strcpy(cmd.url, MAL_URL);
31     }
32   }
33   if (!is_inVM()) {
34     InternetConnect(..., cmd.url, ...);
35     InternetReadFile(..., msg, LEN, ...);
36     fill_command(cmd, msg);
37     if (cmd.op == OP_DOWNLOAD)
38       do_download(cmd.arg);
39     else
40       do_execute(cmd.arg);
41   }
42   else
43     InternetOpenUrl(cmd.url);
44 }
```

Figure 1: Motivating example.

pointer has a very large chance to fall into the pre-allocated region and semantically unrelated invalid pointer dereferences are highly likely to access disjoint pre-allocated regions. In this way, PMP avoids state corruptions with probabilistic guarantees.

While significant effort has been dedicated to crash-free execution, less attention has been given to path exploration, another important aspect that influences the analysis result. In essence, path exploration is a search process that aims to traverse different parts of the subject binary. Since the search space of all possible paths is extremely large for real-world binaries, the state-of-the-art forced execution studies primarily depend on a linear search algorithm [40, 52]. The objective is to cover control flow graph edges by progressively switching additional/different branch outcomes based on the results of previous iterations.

Although simple and effective, linear search has inherent limitations. On one hand, *it may explore a considerable number of unnecessary paths* that do not introduce new program behaviors compared with previous iterations. On the other hand, *it may capture incomplete program behaviors* because certain paths remain unexplored. The fundamental reason is that linear search forcefully alters control flow without considering data flow, potentially missing critical data-flow or inducing infeasible data-flow, both affecting the final exposed malware behaviors. For example, corrupted data-flow may lead to incorrect system call parameters critical for understanding malware behaviors. For better forced execution, a *data-flow aware* path exploration strategy is imperative.

Since define-use relations are crucial for data flow, it is natural to leverage them as a guidance for path exploration. A straightforward way is to utilize the existing dependence analysis approaches (e.g., BDA [54] and VSA [18]) to identify a set of define-use relations and force the program paths to cover the identified ones. Unfortunately, these approaches are conservative, generating a large number of bogus define-use relations that misguide the path exploration. The issues of unnecessary path exploration and incomplete behavior exposure would be even worse than linear search.

In this paper, we propose a novel and practical path exploration strategy, called DueForce (define-use guided path exploration for forced execution), which focuses on the coverage of define-use relations (instead of control flow graph edges) in the subject binary. Rather than utilizing conservative dependence analysis, DueForce generates the define-use relations in a *progressive and self-supervised* way. Specifically, DueForce fuzzes the subject binary by changing the forced branches in the executor. During the fuzzing process, DueForce collects the exercised define-use relations for predicting new ones to cover. The predicted ones are then used as guidance for further exploration.

To assess DueForce's effectiveness, we evaluate it with the SPEC2000 benchmark suite (12 programs of different scales) and 200 recent real-world malware samples (100 on Linux and 100 on Windows). The evaluation results demonstrate that DueForce significantly improves the analysis outcomes of forced execution. Compared to PMP, DueForce achieves a 25.78% higher recall in memory dependence detection and 44.16% higher rate of useful executions, and can expose 167.50% more malicious behaviors for malware analysis. Compared to the path exploration using conservative analysis, DueForce achieves a 305.89% higher recall in memory dependence detection and 331.02% higher rate of useful executions, and can expose 174.76% more malicious behaviors.

This paper makes the following contributions:

- We identify the limitations of path exploration strategy commonly used in existing forced execution engines. These limitations are illustrated through a motivating example simplified from a real-world malware sample.
- We propose a novel and practical define-use guided path exploration strategy for better forced execution. Following the define-use guidance, we can effectively overcome the aforementioned limitations.
- We implement a prototype system and evaluate its effectiveness. The experimental data and source code are available at https://github.com/DueForce/DueForce.

2 MOTIVATION

We use an example to illustrate limitations of linear search and motivate the idea of DueForce. Figure 1 simulates malicious behaviors of a real-world command and control (C&C) malware sample [9].

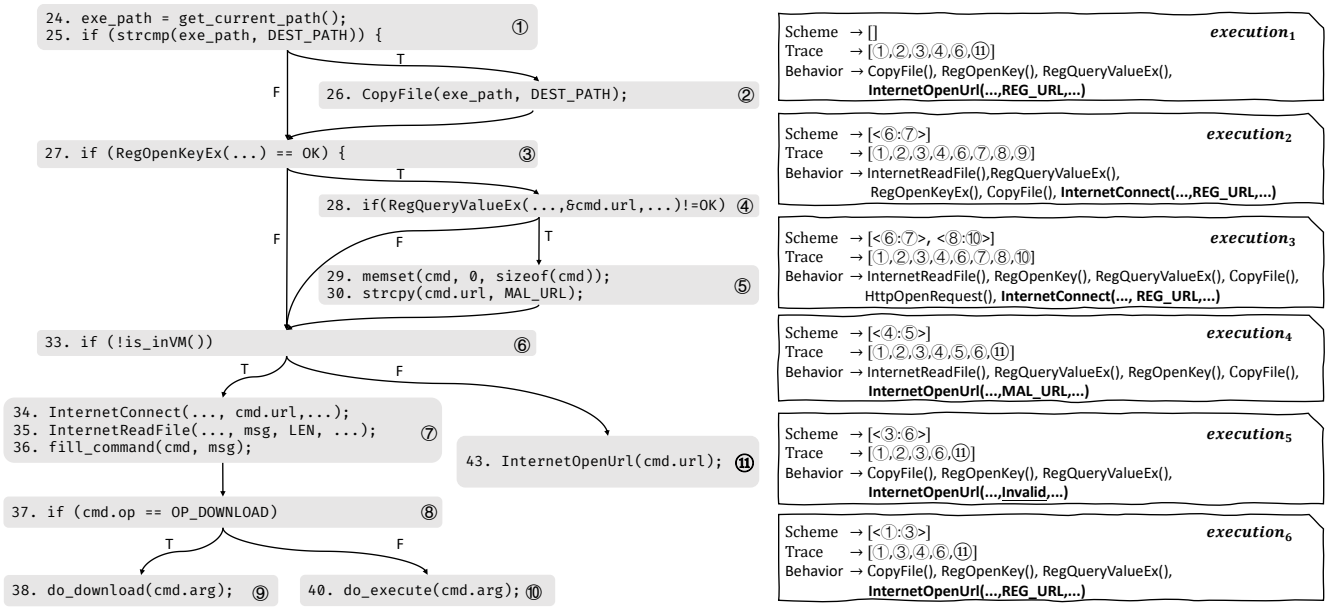


Figure 2: Limitations of the linear search strategy.

The malware disguises itself as an upgrade package of the famous ESET anti-virus software [28]. When executed, it replicates itself to a designated roaming folder (lines 24-26). It then checks for the existence of a special value (line 28) within a specified key (line 27) in the registry, which indicates the presence of ESET on the victim machine. If ESET is not installed, a `Command` object is initialized (lines 29-30), recording the url of the malicious remote server (`MAL_URL`). Otherwise, the url is assigned a regular value (`REG_URL`) extracted from the registry key. The malware is equipped with a VM detector (line 33). Only when it runs on a real machine, will it establish a connection to the C&C server (line 34), receive command messages (lines 35-36), and perform the corresponding actions (line 37-40). Otherwise, it opens the resource specified by the url (line 43). Note that our description is based on source code level for easier comprehension, the analysis is actually operated at binary level.

Simply running the malware sample in a sandbox environment cannot fully expose all of its malicious payloads, since they are safeguarded by highly specific conditions. In our example, if ESET is installed on the victim machine or the malware sample is executed on a virtual machine, the malicious C&C behavior will not be exposed at all. Manually configuring the trigger conditions is time-consuming and not practical for zero-day malware samples. Forced execution provides a systematic solution to explore different program behaviors without environment setup. It works by force-setting a small set of branch outcomes. In this example, if predicates at lines 28, 33, 37 are all forced to take the true branch (while allowing the other predicates to be evaluated as usual), the malicious download behavior is revealed. However, the existing forced execution methods X-Force [40] and PMP [52] fail to find such a path as they employ a simple linear path exploration algorithm.

2.1 Limitations of Linear Search

Linear search aims to cover all control flow edges within several rounds. In each round, a sequence of branches are enforced to direct the control flow from a specified source block to a specified destination block, and the rest branches are evaluated as usual. Such a sequence is called *path scheme*. After each round, a new branch is selected for enforcement, usually by modifying the last branch in the path trace to direct the control flow from its source block to an alternative destination block. Figure 2 shows partial control flow graph of the motivating example and all paths generated by linear search. Assume the malware is executing in a virtual machine with ESET installed. In the initial execution $exection_1$, blocks ① and ③ take the true branch while blocks ④ and ⑥ take the false branch. The path trace covers blocks ①, ②, ③, ④, ⑥, ⑪. The behaviors related to registry, file, and internet operations are disclosed. Since block ⑥ is the source block of the last branch in the path trace and its true branch remains uncovered, linear search appends the branch from block ⑥ to block ⑦ (denoted as ⟨⑥: ⑦⟩) to the path scheme. In the second execution $exection_2$, the path scheme is enforced, resulting in a path trace that covers ①, ②, ③, ④, ⑥, ⑦, ⑧, ⑨. At this time, the branch ⟨⑧: ⑩⟩ is appended to the path scheme for further enforcement. Following the similar manner, after linear search terminates, all control flow edges are covered.

Unnecessary path exploration. Among the six executions, the last two ($execution_5$ and $execution_6$) are unnecessary. In particular, $execution_6$ does not reveal any new behaviors, whether they are different syscalls or the same syscall with different parameter values compared to previous executions. The behaviors occur in $execution_6$ are covered in $execution_1$. Although $execution_5$ reveals the syscall `InternetOpenUrl` with a previously undisclosed parameter value (sourced from `cmd.url`), the value is invalid as it lacks proper initialization. Indeed, both $exeuction_5$ and $execution_6$ do not reveal any new define-use relations.

Incomplete behavior exposure. The linear search does not expose the behavior that triggers the `InternetConnect` syscall with the value `MAL_URL` for its `server_name` parameter. Even though the

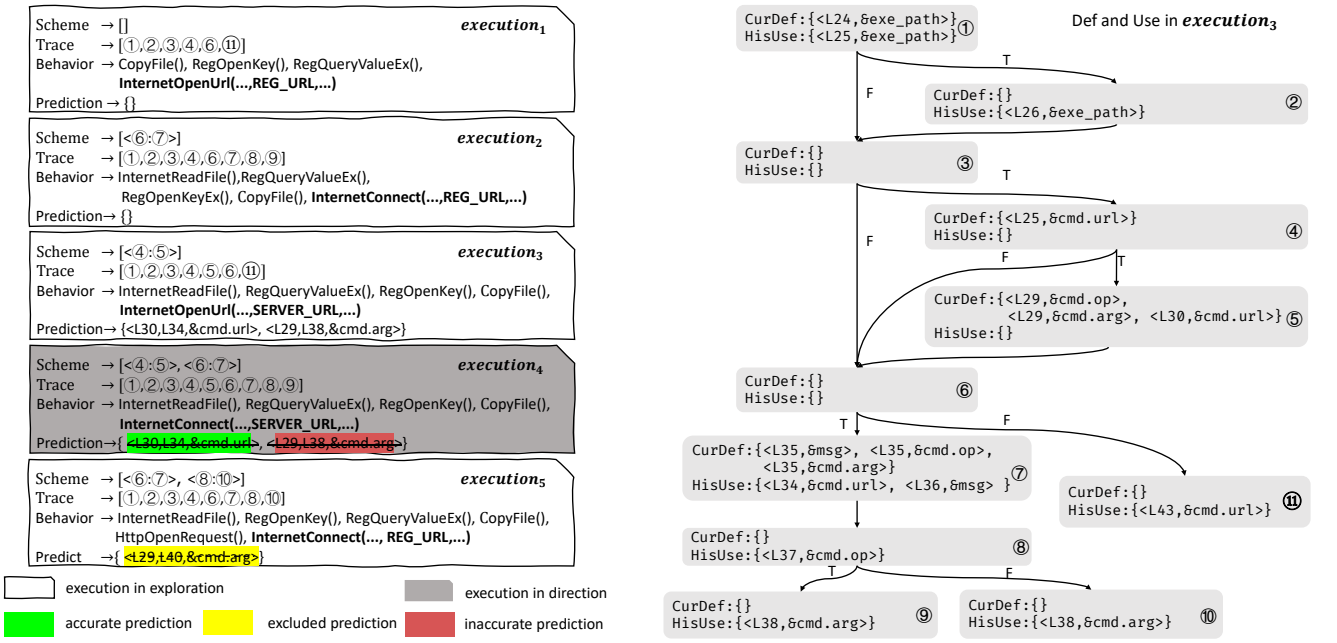


Figure 3: Executions of DueForce on motivating example.

`InternetConnect` syscall was revealed in $execution_3$, its parameter value is `REG_URL`. The behavior that connects to a malicious url (rather than a regular one) is very important for malware analysis. Paths that expose such behavior should contain the sub-trace "④→⑤→⑥→⑦". The linear search fails to generate such paths, regardless of the branch it progressively selects to force in each round. It is unavoidable for linear search that certain control flow edges within the critical sub-path have been covered in previous rounds, thus hindering the traversal of the critical sub-path.

2.2 Observations

Linear search alters control flow without considering data flow, potentially missing critical data-flow (i.e., *under-approximation*) or inducing infeasible data-flow (i.e., *over-approximation*). Intuitively, given a program with n statements, the number of control flow edges could be $O(n^2)$, assuming each statement is capable of transferring control flow to any other; the number of paths could be $O(2^n)$, assuming all branching statements have only two branches; the number of define-use relations could be $O(n^2)$, assuming each statement exhibits data dependence on any other. Hence, a define-use relation may be exposed by many paths, allowing the sampling of paths to cover the majority. Linear search provides a way of path sampling by progressively traversing control flow edges. However, covering all control flow edges does not ensure the revelation of all define-use relations.

There are studies utilizing conservative analysis to detect data dependence. For example, BDA [54] proposes unbiased whole-program path sampling and per-path abstract interpretation for dependence analysis. A natural idea is to use BDA to identify a set of define-use relations and force program paths to cover the identified ones. Unfortunately, the conservative analysis result includes a high number of false positives. When using it as guidance for path exploration, significant time is wasted attempting to cover false dependence. The experiments (Section 4) demonstrate that such an approach performs even worse than linear search. This inspires us to consider a solution that dynamically detects define-use relations and concurrently employs them as guidance for path exploration.

2.3 Our Technique

We develop a fuzzing-based approach to perform path exploration, called DueForce. Unlike traditional fuzzers, DueForce fuzzes the subject binary (instead of input) by changing the forced branches in the executor. There are two modes: the exploration mode that employs control flow coverage guided strategy and the direction mode that employs define-use directed strategy. The two modes are switched under certain plan. After each iteration, DueForce predicts new define-use relations and corrects previous predictions.

Figure 3 illustrates how DueForce works on the motivating example. The executions enclosed in a white and grey background indicate they are in the exploration mode and the direction mode, respectively. The prediction items highlighted with green, red and yellow are accurate, inaccurate and excluded (for correction), respectively. The first two executions ($execution_1$ and $execution_2$) of DueForce are the same with those of linear search. In $execution_3$, DueForce (whose exploration mode focuses on block coverage) and linear search (that focuses on edge coverage) select different branches to force. The executions yield two predicted define-use relations, resulting in mode switching from exploration to direction. One of the predicted item specifies the define-use relation between the statement at line 30 (block ⑤) and that at line 34 (block ⑦) on the `cmd.url` variable. The two blocks appear in the crucial sub-trace that triggers the malicious behavior. The search is guided toward the paths that are anticipated to cover the predicted items.

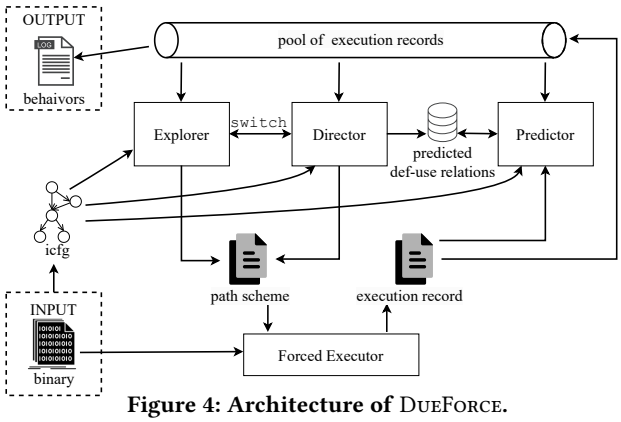


Figure 4: Architecture of DueForce.

In $execution_4$, the behavior that connects to a malicious url is exposed. Section 3 details how exploration, direction, and prediction work. As we can see, none of the five executions of DueForce are unnecessary; the program behaviors exposed by DueForce are more complete than linear search. Our experiment result shows that DueForce has 180.91% higher rate of useful executions and exposes 41.71% more behaviors than linear search on this example.

3 DESIGN

DueForce performs path exploration guided by define-use relations in a progressive and self-supervised way. The architecture is shown in Figure 4. It takes as input a subject binary and its inter-procedural control-flow graph (ICFG), and outputs the program behaviors. It consists of four components: explorer, director, predictor, and forced executor. The explorer and director use different strategies to generate path schemes. In particular, the explorer endeavors to discover new define-use relations by traversing unexplored ICFG blocks, while the director prioritizes covering the predicted define-use relations. The two entail different challenges because executions covering the same ICFG blocks may exercise different sets of define-use relations, depending on variable values. The forced executor runs the subject binary, forcefully setting the respective branch outcomes to enable traversal of the ICFG edges specified in the generated path schemes. The execution records are pooled, so that the predictor can use them to predict new define-use relations for further guidance.

DueForce's overall architecture resembles a fuzzer's. The difference is that the subject binary (instead of input) is fuzzed by changing the forced branches in the executor. The workflow in shown in Algorithm 1. In each iteration, the selection between exploration and direction modes relies on the mode switching strategy (line 4). We provide two strategies: *default* and *lazy*. The default strategy switches between the two modes based on the quantity of unrevealed predicted define-use relations. The lazy strategy shifts to direction mode only when the exploration mode exhausts its ability to cover new ICFG blocks. Both modes will generate a path scheme that specifies the branches requiring forced setting. After forced execution (line 8), the current and historical execution records are used to predict define-use relations to be uncovered (line 9). If any new define-use relation is revealed, the execution record will be pooled for selection in subsequent iterations (lines 10-11). The fuzzing process terminates upon reaching any resource limit, such as time or memory constraints (lines 12-13). Program behaviors are finally extracted from execution records (line 14).

```
                                Block ::= [InsnAddr]
                                 Edge ::= Block × Block
                                 ICFG ::= Block × Edge
                           PathScheme ::= [Edge]
                            PathTrace ::= [Block]
                           ExecRecord ::= PathScheme × PathTrace × Behaivors
                            Def | Use ::= InsnAddr × MemAddr
                               DefUse ::= InsnAddr × InsnAddr × MemAddr
                             Insn2Block ::= InsnAddr ⟶ Block
GetSuccessorSet | GetDescendantSet ::= ICFG × Block ⟶ {Block}
   GetTracePrefix | GetTraceSuffix ::= PathTrace × Block ⟶ PathTrace
                 GetLastForcedBlock ::= PathScheme ⟶ Block
            GetDefSetInTraceBlock ::= PathTrace × Block ⟶ {Def}
            GetUseSetInTraceBlock ::= PathTrace × Block ⟶ {Use}
              GetDefUseSetInTrace ::= PathTrace ⟶ {DefUse}
         GetDefSetInRecordsBlock ::= {ExecRecord} × Block ⟶ {Def}
         GetUseSetInRecordsBlock ::= {ExecRecord} × Block ⟶ {Use}
           GetDefUseSetInRecords ::= {ExecRecord} ⟶ {DefUse}
            GetBlockSetInRecords ::= {ExecRecord} ⟶ {Block}
              EstimateDefUseSet ::= {Def} × {Use} ⟶ {DefUse}
```

Figure 5: Definitions.

3.1 Definitions

To facilitate discussion, we introduce the following definitions as shown in Figure 5. An inter-procedural control flow graph `ICFG` consists of blocks (denoted as `Block`) that are connected by edges (denoted as `Edge`). An ICFG block represents a contiguous sequence of instructions executed without any branching or interruption. An ICFG edge represents the control flow from one ICFG block to another. A path scheme `PathScheme` is defined as a selected set of ICFG edges that require forcefully traversing, and a path trace `PathTrace` is defined as a sequence of ICFG blocks ordered according to their occurrences during program execution. An execution record `ExecRecord` documents the path scheme used in a forced execution, the resulting path trace, and the program behaviors. The define (denoted as `Def`) and use (denoted as `Use`) operations are represented by the instruction address and the memory address that the instruction writes to or reads from. If an instruction $insn_1$ writes to a memory address *mem*, which is subsequently read by another instruction $insn_2$, then $isns_1$ and $insn_2$ have a define-use relation (denoted as `DefUse`) at memory *mem*.

We define some utility functions for brief description. `Insn2Block` maps an instruction to its containing ICFG block. `GetSuccessorSet` and `GetDescendantSet` respectively obtain the successors and descendants of a given ICFG block. `GetTracePrefix` and `GetTraceSuffix` respectively extract the sub-trace that concludes and commences with a given ICFG block from a path trace. `GetLastForcedBlock` locates the ICFG block whose branch outcome requires forcefully set to enable traversal of the last ICFG edge specified in a given path scheme. `GetDefSetInTraceBlock`, `GetUseSetInTraceBlock`, `GetDefUseSetInTrace` respectively retrieve the define and use operations and the define-use relations that have taken place within (a given ICFG block of) a provided path trace. Likewise, `GetDefSetInRecordsBlock`, `GetUseSetInRecordsBlock`, `GetDefUseSetInRecords` retrieve the define and use information from the historical execution records. `GetBlockSetInRecords` returns the ICFG blocks that have already been covered in previous executions. `EstimateDefUseSet` estimates define-use relations for the given sets of define and use operations.

Algorithm 1: Overall workflow.

```
Input  : binary                          ▷ subject binary
         icfg                            ▷ ICFG of the subject binary
Output : behaviors                       ▷ program behaviors
Global : records                         ▷ execution records
         predicted_dus                   ▷ predicted define-use relations
         pm                              ▷ prediction metric
         ps                              ▷ program states
1  records ← ∅, predicted_dus ← ∅
2  pm ← {}, ps ← {}
3  while True do
4      if ModeSwichingStrategy(predicted_dus) == EXPLORE then
5          scheme_new ← explore(icfg, records)
6      else
7          scheme_new ← direct(icfg, records, predicted_dus)
8      record ← ForceExecute(binary, sheme_new)
9      ⟨predicted_dus, pm, ps⟩ ← predict(icfg, records, record, predicted_dus, pm, ps)
10     if HasNewDefUse(records, record) then
11         records ← records ⋃ record
12     if ReachResourceLimit() then
13         break
14 behaviors ← ExtractBehaviorsFromRecords(records)
```

3.2 Exploration

In the exploration mode, we employ control flow coverage guided fuzzing strategy. Intuitively, it tries to cover the ICFG blocks by using as shortest path schemes as possible. Note that the fewer branch outcomes that are set forcefully, the more closely the forced execution resembles a normal execution. This allows us to avoid bogus define-use relations as much as possible.

Algorithm 2 describes how the exploration mode works. It first extracts previously covered ICFG blocks from the historical execution records (line 2). It then searches for the force candidates for each path scheme within the historical execution records, aiming to identify those that will lead to the exploration of uncovered ICFG blocks subsequent to the last forced ICFG block in the path scheme (lines 4-11). It selects the shortest path scheme that has at least one associated force candidate (lines 12-14). For the selected path scheme, the mutation is performed by appending it with a randomly selected force candidate (lines 15-17).

Example. Take the executions of the motivating example as an illustration (Figure 3). The initial execution $execution_1$ uses an empty path scheme, the resulting path trace covers the ICFG blocks numbered ①,②,③,④,⑥,⑪. In $execution_2$, the empty path scheme is selected for mutation, as it is the (sole) shortest one in the historical execution records. The force candidates include ⟨④: ⑤⟩ and ⟨⑥: ⑦⟩, since they will lead to the exploration of uncovered ICFG blocks numbered ⑤ and ⑦, respectively. The latter one is randomly selected and appended to the empty path scheme, generating a mutated path scheme [⟨⑥: ⑦⟩]. The force execution using the mutated path scheme results in the path trace "①→②→③→④→⑥→⑦→⑧→⑨". This execution exposes new define-use relations, hence is stored in the execution record pool. Similarly, in $execution_3$ and $execution_5$, the shortest path scheme that has at least one associated force-setting candidate is selected for mutation; and the executions are stored, as they expose new define-use relations.

3.3 Direction

In the direction mode, we employ define-use directed fuzzing strategy. Intuitively, it guides the search towards the paths that are anticipated to expose more previously uncovered define-use relations. Given a path scheme randomly selected from the historical execution records, we will weight all the force candidates based on their potential in revealing predicted define-use relations.

Algorithm 3 describes how the direction mode works. It randomly selects a historical execution record (line 2), locates the last forced block in the path scheme (line 3), and extracts the sub-trace that commences with the located block from the path trace (line 4). For each block in the sub-trace, we retrieve the define operations occurred prior to the block in the path trace (lines 6-10) and examine each of its succeeding blocks (lines 11-21). The examination retrieves the use operations occurred in the descendants of the succeeding block within the historical execution records (lines 15-19), estimates the expected define-use relations that might emerge if the succeeding block is forcefully traversed (line 20), and assigns a weight to the force candidate based on the size of the intersection of the expected define-use relations and the predicted ones (line 21). Finally, we sample a force candidate based on the assigned weights (line 22) to generate a new path scheme (line 23). The higher the weight, the more likely a force candidate is to be sampled.

Algorithm 2: Exploration.

```
Input  : icfg:          ICFG
         records:       {ExecRecord}
Output : scheme_new:    PathScheme
1  scheme_selected ← ∅, force_candidates ← ∅
2  cov_blocks ← GetBlockSetInRecords(records)
3  for ⟨scheme, trace, −⟩ in records do
4      tmp_candidates ← ∅
5      last_forced_block ← GetLastForcedBlock(scheme)
6      trace_suffix ← GetTraceSuffix(trace, last_forced_block)
7      for suffix in trace_suffix do
8          successors ← GetSuccessorSet(icfg, suffix)
9          for succ in successors do
10             if succ ∉ cov_blocks then
11                 tmp_candidates ← tmp_candidates ⋃ ⟨suffix, succ⟩
12     if tmp_candidates ≠ ∅ and Len(scheme) ≤ Len(scheme_selected) then
13         scheme_selected ← scheme
14         force_candidates ← tmp_candidates
15 if scheme_selected ≠ ∅ then
16     force_selected ← Random(force_candidates)
17     scheme_new ← Append(scheme_selected, force_selected)
18 return scheme_new
```

Example. Refer to the left part of Figure 3 for the executions of the motivating example. After three executions, the number of predicted items exceeds the threshold (2 in this simplified example), DueForce switches to the direction mode. Assume $execution_3$ is randomly selected for consideration, the path trace is "① → ②→③→④→⑤→⑥→⑪". The force candidates include ⟨①: ③⟩, ⟨③: ⑥⟩, ⟨④: ⑥⟩, and ⟨⑥: ⑦⟩. Take ⟨⑥: ⑦⟩ as an example. Refer to the right part of Figure 3, the define operations occurred prior block ⑥ in the path trace include ⟨$L24$, `&exe_path`⟩, ⟨$L29$, `&cmd.op`⟩, ⟨$L29$, `&cmd.arg`⟩, and ⟨$L30$, `&cmd.url`⟩; the use operations occurred in the descendants of block ⑦ within the historical execution records include ⟨$L34$, `&cmd.url`⟩, ⟨$L36$, `&msg`⟩, and ⟨$L38$, `&cmd.arg`⟩. The expected define-use relations if ⟨⑥: ⑦⟩ is forcefully taken are {⟨$L30$, $L34$, `&cmd.url`⟩, ⟨$L29$, $L38$, `&cmd.arg`⟩} (by merging the defines and uses that operate on the same memory address). The expected define-use relations match the predicted ones; hence the

Algorithm 3: Direction.

```
Input  : icfg:          ICFG
         records:       {ExecRecord}
         predicted_dus: {DefUse}
Output : scheme_new:    PathScheme
1  force_candidates ← ∅,  weights ← {}
2  ⟨scheme_selected, trace_selected, −⟩ ← Random(records)
3  last_forced_block ← GetLastForcedBlock(scheme_selected)
4  trace_suffix ← GetTraceSuffix(trace_selected, last_forced_block)
5  for suffix in trace_suffix do
6      defs ← ∅
7      trace_prefix ← GetTracePrefix(trace_selected, suffix)
8      for prefix in trace_prefix do
9          tmp_defs ← GetDefSetInTrace(trace, prefix)
10         defs ← defs ⋃ tmp_defs
11     successors ← GetSuccessorSet(icfg, suffix)
12     for succ in successors do
13         force ← ⟨suffix, succ⟩
14         force_candidates ← force_candidates ⋃ force
15         uses ← ∅
16         descendants ← GetDescendantSet(icfg, succ)
17         for desc in descendants do
18             tmp_uses ← GetUseSetInRecordsBlock(records, desc)
19             uses ← uses ⋃ tmp_uses
20         expected_dus ← EstimateDefUseSet(defs, uses)
21         weights[force] ← Size(expected_dus ⋂ predicted_dus)
22 force_selected ← WeightedRandom(force_candidates, weights)
23 scheme_new ← Append(scheme_selected, force_selected)
24 return scheme_new
```

weight of ⟨⑥ : ⑦⟩ is designated as 2, reflecting the size of the intersection. The weights of other force candidates are calculated using the same method. Out of all the force candidates, ⟨⑥ : ⑦⟩ is the most probable choice for generating new path scheme, as it possesses the highest weight.

3.4 Prediction

The objective of predication is to generate new define-use relations (providing as guidance for path exploration) based on previously disclosed define-use relations. Intuitively, the define and use operations occurred in the paths collected from individual historical execution records are propagated through all paths to predict new define-use relations. Specifically, it computes program states that represent the set of live memory addresses at each ICFG block and their definition instructions. An intuitive correspondence at the source level is the set of live variables at a program point and the statements that define them.

Define-use relations are predicted between a read (use) instruction and all the definitions (write instructions) to the memory address being read. The feedback from each execution instance is gathered to assess the efficacy of the prediction. The memory addresses corresponding to accurately predicted define-use relations will be rewarded, whereas those corresponding to inaccurate ones will face penalties. The higher the association of a memory address with accurate predictions, the greater its likelihood of being used in future prediction. This allows generating define-use relations in a progressive and self-supervised way. Compared to utilizing the results of conservative dependence analysis, on-the-fly prediction with correction provides more effective guidance, resulting in superior performance.

Algorithm 4 presents the details of prediction. It takes as input the inter-procedural control flow graph (*icfg*), the current and historical execution records (*record* and *records*), the predicted

Algorithm 4: Prediction.

```
Input  : icfg:           ICFG
         records:        {ExecRecord}
         record:         ExecRecord
         predicted_dus:  {DefUse}
         pm:             MemAddr → ⟨Int,Int,Int⟩
         ps:             Block → (MemAddr → {Block})
Output : predicted_dus': {DefUse}
         pm':            MemAddr → ⟨Int,Int,Int⟩
         ps':            Block → (MemAddr → {Block})
1  predicted_dus' ← predicted_dus,  pm' ← pm,  ps' ← ps
2  ⟨scheme, trace, −⟩ ← record
3  cur_dus ← GetDefUseSetInTrace(trace)
4  for ⟨insn_def, insn_use, mem⟩ in predicted_dus' do
5      block_def ← Insn2Block(insn_def)
6      block_use ← Insn2Block(insn_use)
7      ⟨accurate_cnt, inaccurate_cnt, total_cnt⟩ ← pm'[mem]
8      if ⟨insn_def, insn_use, mem⟩ ∈ cur_dus then
9          accurate_cnt ← accurate_cnt + 1
10         predicted_dus' ← predicted_dus' \ {⟨insn_def, insn_use, mem⟩}
11     else if block_def ∈ trace and block_use ∈ trace then
12         inaccurate_cnt ← inaccurate_cnt + 1
13         predicted_dus' ← predicted_dus' \ {⟨insn_def, insn_use, mem⟩}
14     total_cnt ← total_cnt + 1
15     pm'[mem] ← ⟨accurate_cnt, inaccurate_cnt, total_cnt⟩
16 for block in trace do
17     his_defs ← GetDefSetInRecordsBlock(records, block)
18     cur_defs ← GetDefSetInTraceBlock(trace, block)
19     new_defs ← cur_defs \ his_defs
20     if new_defs ≠ ∅ then
21         mem_addrs ← ∅
22         for ⟨insn, mem⟩ in new_defs do
23             mem_addrs ← mem_addrs ⋃ mem
24         mem_addrs ← WeightedSample(mem_addrs, pm')
25         descendants ← GetDescendantSet(icfg, block)
26         for desc in descendants do
27             for mem in mem_addrs do
28                 ps'[desc][mem] ← ps'[desc][mem] ⋃ block
29     his_uses ← GetUseSetInRecordsBlock(records, block)
30     cur_uses ← GetUseSetInTraceBlock(trace, block)
31     all_uses ← cur_uses ⋃ his_uses
32     for ⟨insn, mem⟩ in all_uses do
33         for block_def in ps'[block][mem] do
34             predicted_dus' ← predicted_dus' ⋃ ⟨block_def, block⟩
35 return ⟨predicted_dus', ps', pm'⟩
```

define-use relations (*predicted_dus*), the metric of the prediction (*pm*), and the program states (*ps*). After processing, it updates the predicted define-use relations (*predicted_dus'*), the metric of prediction (*pm'*), and the program states (*ps'*). The process consists of two stages. In the first stage (lines 4-15), each predicted define-use relation is assessed based on the feedback of current execution record. Specifically, a predicted item will be considered as accurate, if it appears within the define-use relations extracted from the current execution trace (lines 8-10); otherwise, if a predicted item does not appear but its define and use ICFG blocks are present within the current execution trace, it will be considered as inaccurate (lines 11-13). The accurate and inaccurate predicted items are removed from the predicted results (lines 10 and 13). The assessment is used to update the metric of prediction (line 15). In the second stage (lines 16-34), the ICFG is traversed to compute program states of each block for incremental prediction. Specifically, newly occurred define operations (*new_defs*) are extracted from the current execution trace (lines 17-19) and then propagated to the descendants of the block where the define operations occur (lines 20-28). For each ICFG block in the current execution trace, all use operations occurred in the block (*all_uses*) are extracted from both the current

and historical execution records (lines 29-31). Define-use operations are predicted by finding memory addresses that are written by any item in *new_defs* and read by any item in *all_uses* (lines 32-34). The rewards or penalties assigned to a memory address influence its likelihood of being sampled for prediction (line 24).

Example. Recall the executions of the motivating example. As shown in the right part of Figure 3, after $execution_3$, the newly occurred define operations in block ⑤ include ⟨$L30$, `&cmd.url`⟩, ⟨$L29$, `&cmd.arg`⟩, and ⟨$L29$, `&cmd.op`⟩; and all use operations occurred in block ⑦ and block ⑨ include ⟨$L34$, `&cmd.url`⟩,⟨$L36$, `&msg`⟩ and ⟨$L38$, `&cmd.arg`⟩, respectively. By merging the new defines in block ⑤ with all uses in block ⑦ and block ⑨, we predict two define-use relations: ⟨$L30$, $L34$, `&cmd.url`⟩ and ⟨$L29$, $L38$, `&cmd.arg`⟩, respectively. Then in $execution_4$, ⟨$L30$, $L34$, `&cmd.url`⟩ appears in the define-use relations extracted from the execution, hence is treated as accurate; ⟨$L29$, $L38$, `&cmd.arg`⟩ does not appear but its define and use blocks (block ⑤ and block ⑨) are present in the execution trace, hence is treated as inaccurate. In $execution_5$, the new prediction ⟨$L29$, $L40$, `&cmd.arg`⟩ is most likely to be excluded as a penalty on `cmd.arg`.

3.5 Implementation

Forced Executor. The forced executor of DueForce is a reimplementation of PMP's [52] in C/C++. PMP originally builds its forced executor is built on top of QEMU user-mode emulator [12]. Unlike system-wide emulation, QEMU currently only supports user space emulation on Linux. For analyzing malware samples on both Linux and Windows, we re-implement the forced executor implemented on top of Pin [11], a well-known dynamic binary instrumentation tool, in order to handle both Linux and Windows malware samples. Note that previous forced executors such as PMP [52] were implemented in QEMU [12] and could not handle Windows malware.

Loop and Recursion. As forced execution may lead to infinite loop and infinite recursion, it's necessary to handle these issues to enable the execution to continue. Both PMP and X-Force reset the loop bound and callstack depth to a pre-defined constant. To handle infinite recursion, DueForce employs the same technique as PMP. To handle infinite loop, DueForce set loop bound similarly. However, due to the limitation of linear search, loop analysis occurs only once, and forcing the program to enter the loop every time consumes excessive time. To this end, DueForce determines whether to enter into the loop on demand. When predicted define-use relationships exist within the loop, DueForce enforces the control flow into the loop even if the loop body have been covered previously.

4 EVALUATION

We evaluate DueForce on the SPEC2000 benchmark suite [17] and a set of recent malware samples collected from VirusShare [7]. The experiments on SPEC2000 are conducted on a desktop computer equipped with 8-core CPU (Intel i7-6700 @ 4.00GHz) and 32GB memory. The experiments on the malware samples are conducted on a virtual machine (for sand-boxing their malicious behaviours) hosted on the same desktop. We compare DueForce with PMP [52] and the approach that uses the dependence analysis results of BDA [54] to guide path exploration (referred to as BDA* in the following description), as mentioned in Section 2.2. Note that PMP has a finite number of executions (approximately equivalent to the number of control flow edges), whereas DueForce and BDA* run in a infinite loop. For a fair comparison, we enhance PMP to incorporate random path exploration following the completion of its linear search. This enables us to compare the three tools in the same time duration.

4.1 SPEC2000 Analysis

SPEC2000 is a benchmark suite designed to evaluate the performance of computer processors. It contains a series of real world programs of various scale (ranging from 6K to 4M) that cover a wide spectrum of computing tasks. We evaluate the execution outcomes and the memory dependence (i.e., define-use relations). For each program, we first record the number of rounds in which PMP completes its linear search; then we run DueForce for the equivalent number of rounds and record the time consumption; and finally we keep running PMP and BDA* for the same time duration. We also conduct an ablation study that configures DueForce with different settings, including the exploration mode alone, the combination of exploration and direction without self-supervision, and all components enabled but utilizing lazy mode switching. Besides, the results are averaged over five trials.

For execution outcomes, we measure the rate of useful executions. An execution is considered useful if it reveals previously uncovered define-use relations. For memory dependence, it is intractable to acquire the ground truth, even with source code (due to various reasons such as aliasing and loops). We use two methods (as done in PMP and BDA) to evaluate the quality of detected dependencies. First, we run the programs with the inputs provided by SPEC2000 and use the observed dependencies as reference. Any dependence that appears in the reference but not in the detected results is a *missing* one (or a false negative). Second, we use a static type checker to verify the detected dependencies. Any detected dependence whose source and destination have different types is considered a *mistyped* one (which must be a false positive).

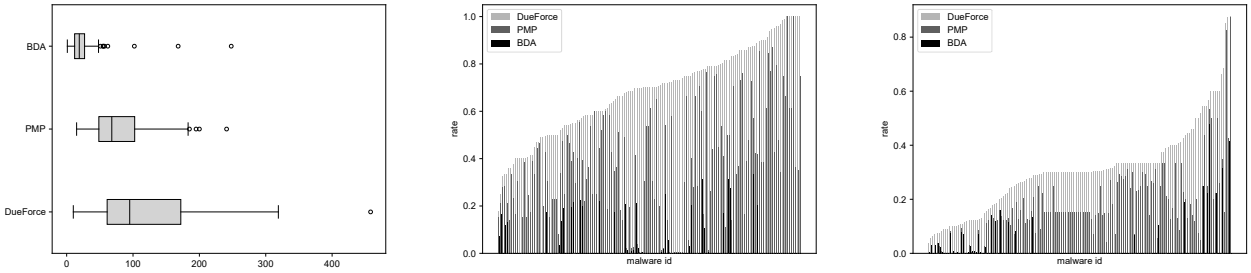
Overall results. Table 1 presents the overall results. Column 1 is the program name. Column 2 reports the execution time. Column 3 denotes the number of dependencies observed in the reference execution. Columns 4-7, 8-11 and 12-15 are the results of DueForce, PMP and BDA*, respectively. The columns labeled with "# Found", "# Correct" and "# Mistyped" report the numbers of detected, correctly detected and mistyped dependencies, respectively. The columns labeled with "# Execution" report the numbers of useful and total executions. We have the following observations. (1) DueForce exhibits the highest rate of useful executions, surpassing PMP and BDA* by an average of 44.16% and 331.02%, respectively. (2) The average rate of correctly detected memory dependence (recall) reported by DueForce is 25.78% and 305.89% higher than those of PMP and BDA*, respectively. (3) DueForce exhibits a comparable average rate of mistyped memory dependence (false positives) to those of PMP and BDA*. The experiment results align with the expectation. PMP exhibits lower rate of useful executions compared to DueForce, hence it misses more memory dependence than DueForce. BDA* utilizes the conservative memory dependence result as guidance for path exploration; significant time is wasted attempting to cover false define-use relations.

Table 1: Overall results on the SPEC2000 benchmark suite.

Program	Time (s)	# Refer	DueForce				PMP				BDA*			
			# Found	# Correct	# Mistyped	# Execution	# Found	# Correct	# Mistyped	# Execution	# Found	# Correct	# Mistyped	# Execution
164.gzip	192	3,648	3,691	2,754 (75.49%)	13 (0.35%)	259/361 (71.88%)	3,648	2,525 (69.22%)	1 (0.03%)	281/708 (39.69%)	1,689	1,509 (41.37%)	1 (0.06%)	34/321 (10.60%)
175.vpr	741	13,962	13,833	9,567 (68.52%)	260 (1.88%)	790/861 (91.85%)	11,151	7,646 (54.88%)	140 (1.26%)	729/1,089 (66.94%)	1,782	1,049 (7.51%)	9 (0.51%)	138/861 (16.03%)
176.gcc	7,660	324,884	66,270	37,222 (11.00%)	2,715 (4.10%)	7,454/8,000 (93.17%)	46,630	21,804 (6.71%)	1,959 (4.20%)	6,279/1,1171 (56.21%)	10,772	6,975 (2.15%)	317 (2.94%)	675/8,000 (8.44%)
181.mcf	27	2,053	2,097	1,539 (74.96%)	34 (1.68%)	95/111 (85.82%)	1,629	1,237 (60.25%)	92 (5.65%)	72/108 (66.67%)	686	559 (27.23%)	17 (2.48%)	34/121 (28.10%)
186.crafty	4,607	31,631	25,726	16,922 (53.50%)	144 (0.56%)	1,927/2,431 (79.29%)	21,238	14,227 (44.98%)	100 (0.47%)	1,771/4,183 (42.34%)	7,231	4,635 (17.78%)	21 (0.29%)	175/1,671 (10.47%)
197.parser	519	16,575	10,697	8,031 (48.45%)	276 (2.58%)	1,384/1,541 (88.66%)	9,975	6,856 (41.36%)	890 (8.92%)	1,300/2,060 (63.11%)	1,206	894 (5.39%)	14 (1.16%)	91/1,541 (5.91%)
252.eon	470	8,958	9,200	4,222 (47.13%)	83 (0.90%)	442/491 (90.06%)	9,203	4,164 (46.48%)	102 (1.11%)	429/585 (73.33%)	1,132	1,057 (11.80%)	0 (0.00%)	25/481 (5.20%)
253.perlbmk	5,468	61,939	34,575	16,945 (27.36%)	932 (2.69%)	3,305/3,581 (92.31%)	35,021	16,882 (27.26%)	876 (2.50%)	4,113/5,913 (69.56%)	5,276	3,926 (6.34%)	405 (7.68%)	226/3,581 (6.31%)
254.gap	2,008	42,276	7,899	3,635 (8.60%)	225 (2.82%)	761/901 (84.50%)	7,728	2,843 (6.72%)	69 (0.76%)	901/1,559 (57.79%)	1,994	1,190 (2.81%)	58 (2.91%)	105/901 (11.65%)
255.vortex	5,908	42,523	54,828	20,924 (49.21%)	926 (1.69%)	5,368/5,791 (92.71%)	44,804	17,672 (41.56%)	610 (1.36%)	4,544/7,673 (58.44%)	11,861	5,532 (13.01%)	160 (1.35%)	746/5,801 (12.86%)
256.bzip2	522	4,306	3,739	3,155 (73.27%)	17 (0.45%)	163/241 (67.97%)	3,265	2,797 (64.96%)	9 (0.28%)	188/402 (46.77%)	1,587	1,399 (32.49%)	2 (0.13%)	42/261 (16.10%)
300.twolf	1,500	17,876	26,721	11,880 (66.46%)	783 (2.92%)	2,381/2,841 (83.81%)	23,424	10,086 (56.42%)	554 (2.37%)	2,138/3,139 (68.11%)	6,685	4,974 (27.83%)	54 (0.81%)	331/2,901 (11.41%)
Avg.	-	-	-	**11,398.50**	**534.92**	**85.17%**	-	**9,062**	**450.17**	**59.08%**	-	**2,808.25**	**88.17**	**19.76%**

Table 2: Ablation study on the SPEC2000 benchmark suite.

Program	Time (s)	# Refer	Exploration Only				Exploration + Direction (w/o Self-Supervision)				DueForce (lazy mode switching)			
			# Found	# Correct	# Mistyped	# Execution	# Found	# Correct	# Mistyped	# Execution	# Found	# Correct	# Mistyped	# Execution
164.gzip	192	3,648	3,172	2,385 (65.38%)	2 (0.06%)	215/361 (59.57%)	3,148	2,364 (64.80%)	2 (0.06%)	207/361 (57.34%)	3,678	2,774 (76.04%)	21 (0.57%)	271/361 (75.07%)
175.vpr	741	13,962	12,734	8,794 (62.99%)	188 (1.48%)	767/861 (89.08%)	13,460	9,039 (65.16%)	237 (1.76%)	779/861 (90.48%)	13,443	9,267 (66.37%)	270 (1.65%)	776/861 (90.13%)
176.gcc	7,660	324,884	41,778	19,593 (6.03%)	1,652 (3.95%)	5,017/8,001 (62.70%)	40,260	18,560 (5.71%)	1,732 (4.30%)	4,882/8,001 (61.02%)	48,208	21,681 (6.67%)	2,019 (4.19%)	5,743/8,001 (71.78%)
181.mcf	27	2,053	1,918	1,441 (70.19%)	54 (2.87%)	83/111 (74.77%)	1,936	1,473 (71.75%)	68 (3.51%)	87/111 (78.38%)	2,016	1,563 (76.13%)	60 (2.98%)	96/111 (86.49%)
186.crafty	4,607	31,631	21,583	15,077 (47.67%)	89 (0.41%)	1,345/2,431 (55.33%)	21,213	14,790 (46.76%)	91 (0.43%)	1,310/2,431 (53.89%)	24,324	15,680 (49.57%)	95 (0.39%)	1,810/2,431 (74.45%)
197.parser	519	16,575	8,844	6,327 (38.17%)	613 (6.93%)	939/1,541 (60.35%)	9,695	6,619 (39.93%)	778 (8.02%)	1,107/1,541 (71.84%)	10,923	7,630 (46.03%)	990 (9.06%)	1,297/1,541 (84.17%)
252.eon	470	8,958	8,834	3,997 (44.62%)	87 (0.98%)	390/491 (79.43%)	9,069	4,226 (47.18%)	98 (1.08%)	391/491 (79.63%)	9,389	4,381 (48.91%)	81 (0.86%)	405/491 (82.48%)
253.perlbmk	5,468	61,939	21,644	10,695 (17.27%)	1,310 (6.05%)	1,819/3,581 (50.80%)	29,196	15,508 (25.04%)	787 (2.70%)	2,920/3,581 (81.54%)	31,900	15,963 (25.77%)	1,313 (4.12%)	3,369/3,581 (94.08%)
254.gap	2,008	42,476	7,736	2,749 (6.50%)	60 (0.78%)	712/901 (79.02%)	8,358	3,488 (8.25%)	138 (1.65%)	784/901 (87.01%)	8,237	3,712 (8.78%)	208 (2.51%)	825/901 (91.56%)
255.vortex	5,908	42,523	39,634	16,116 (37.90%)	502 (1.27%)	3,808/5,791 (65.76%)	42,991	17,653 (41.51%)	293 (1.18%)	4,181/5,791 (72.15%)	50,524	19,908 (46.82%)	800 (1.58%)	5,211/5,791 (89.98%)
256.bzip2	522	4,306	2,311	2,020 (46.91%)	2 (0.09%)	96/241 (39.83%)	2,936	2,576 (59.82%)	2 (0.07%)	127/241 (52.70%)	3,774	3,180 (73.85%)	21 (0.56%)	161/241 (66.80%)
300.twolf	1,500	17,876	23,345	10,036 (56.14%)	801 (3.43%)	1,958/2,841 (68.92%)	25,475	11,237 (62.86%)	1,177 (4.62%)	2,276/2,841 (80.11%)	25,276	11,691 (65.40%)	710 (2.81%)	2,394/2,841 (84.27%)
Avg.	-	-	-	**8,269.17**	**446.67**	**66.25%**	-	**8,961.08**	**450.25**	**72.30%**	-	**9,785.83**	**549**	**82.55%**



(a) number of exposed syscall sequences. **(b) rate of useful executions (define-use).** **(c) rate of useful executions (behavior).**

Figure 6: Overall results of malware analysis.

```
1  typedef struct {
2    char *string;  /* the word itself */
3    DictNode *left, *right;
4  } DictNode;
5  void insert_word(DictNode *dn, int l) {
6    if (contains_underbar(dn->string))  {
7      new = alloc_idiom_node_from(dn);
8      dn->left = dn->right = NULL;
9      xfree(dn, sizeof (DictNode)); //crash here
10     /* the code in this region will not be covered by linear search */
11   }
12 }
13 void read_word(int i) {
14   DictNode *dn = NULL;
15   if (token[0] == '/') {
16     dn = xalloc(sizeof(DictNode));
17     ...
18   }
19   insert_word(dn, i);
20 }
```

Figure 7: SPEC2000 case study.

Ablation study. Table 2 presents the results of ablation study. In this experiment, the execution time for each program remains consistent with the aforementioned experiment. We have the following observations. (1) The combination of exploration and direction increases the rate of useful executions by 28.57% and increases the rate of correctly detected memory dependence by 37.84%, compared to utilizing the exploration mode alone. (2) Implementing self-supervision, which leverages feedback from each execution instance to rectify predictions, leads to a 17.80% increase in the rate of useful executions and a 27.20% increase in the rate of correctly detected memory dependence. (3) Lazy mode switching strategy, which shifts to direction mode only when the exploration mode exhausts its ability to cover new ICFG blocks, demonstrates performance comparable to the default mode switching strategy.

Case study. We use 197.*parser* as a case study to demonstrate the advantages of DueForce. It is a syntactic parser of English base on a user-defined dictionary. Figure 7 shows the simplified code snippet related to the pre-processing of the dictionary. Specifically, it reads words from a user-defined dictionary (lines 13-20) and inserts them into an internal dictionary tree (lines 5-12). In a plain execution, no user-defined dictionary is provided; both the predicates at lines 15 and 6 take the false branch. Linear search will iteratively force-set the outcomes of predicates 6 and 15 in two different executions. In the execution where predicate at line 6 is forced to take the true branch, the program crashes at line 9 because `dn` is `NULL`, as the predicate at line 15 is not forced to take the true branch. The code highlighted in red remains uncovered after the linear search terminates. While in DueForce, the define-use relation $\langle L16, L7, \texttt{\&dn}\rangle$ is predicted. Guided by such a define-use relation, the path that forcibly takes the true branch at lines 15 and 6 is explored. As a result, the code highlighted in red gets chance to be executed, potentially exposing more define-use relations.

4.2 Malware Analysis

We use 200 malware samples to evaluate DueForce's capability in exposing hidden malicious behaviors. These samples cover recent malware of different families. Half of them are on the Windows platform, and the other half are on the Linux platform. We compare DueForce's analysis results with those of PMP and BDA* on different aspects, including the number of exposed behaviors, the

```
1  struct SocketStruct {
2    int socket, type, usrId, pwdId;
3    char * IP;
4  }S;
5  void main() {
6      char * usr, *pwd;
7      int MainCommSocket = connect_by_ip(ATTACKER_IP);
8      while (1) {
9        if (!MainCommSocket) break;
10       switch (S.type) {
11         case 0 :
12           S.IP = getIntranetIP();
13           usr = usrname[S.usrId];
14           pwd = passwd[S.pwdId];
15           S.socket = connect_by_ip(S.IP);
16           break;
17         case 1 :
18           send(S.socket, usr, strlen(usr));
19           send(S.socket, pwd, strlen(pwd));
20           if (readUntil(S.socket, "correct")) S.type = 2;
21           else modify(&S.usrId, &S.pwdId);
22           break;
23         case 2 :
24           sprintf(msg, "REPORT %s:%s:%s", S.IP, usr, pwd);
25           send(MainCommSocket, msg, strlen(msg));
26         default:
27           break;
28       }
29     }
30 }
```

Figure 8: Malware case study.

rate of useful executions in revealing new define-use relations and exposing new behaviors. We re-implement PMP's forced executor on top of Pin [11] to support cross-platform malware analysis. We failed to port BDA to Windows, hence we only use BDA* for analyzing the Linux malware samples.

Following the standard practices in malware analysis, we capture malware behaviors by monitoring system calls (on Linux) and system library APIs (on Windows) invoked by the malware sample. We refer to system calls and system library APIs as "syscalls" for consistency. Intuitively, the greater the number of syscalls exposed through forced execution, the more malware conditions are triggered. The list of interesting syscalls is provided by Cuckoo [1], a famous sandbox for malware analysis. We treat distinct invocations of the same syscall as separate behaviors, if their arguments exhibit significant differences. Specifically, we do not distinguish between two distinct integer argument values. Regarding two string arguments, we consider them as distinct if their similarity is below 80%. For structure arguments, we determine their dissimilarity by examining discrepancies in their crucial fields (e.g., the `sin_addr` field for the `sockaddr_in` structure). When dealing with pointer arguments, we examine their dereferenced values.

For each malware sample, we initiate a pristine virtual machine snapshot to serve as analysis environment. To mitigate the side effects across multiple executions of a malware sample, we perform light-weight recovery of execution environment. Specifically, we intercept the syscalls that may affect global objects, preserve the current states of the target objects before invoking these syscalls, and subsequently restore their states after an execution instance of the malware sample. For example, we intercept the `open` system call on Linux. If the file is opened in the write mode, we copy the file to a temporary path and restore it post-execution. For fairness, the environment initialization and recovery are equally enforced for DueForce, PMP and BDA*.

Result summary. In our evaluation, DueForce exposes more syscall sequences than PMP and BDA* in 93.5% and 99% of malware samples, respectively. Figure 6 presents the overall results of malware analysis. The detailed results are shown in [10]. Specifically, the number of unique syscall sequences exposed by different tools are shown in Figure 6a. On average, DueForce reports 35.02% and 197.64% more syscall sequences over PMP and BDA*, respectively. The rates of useful executions in revealing new define-use relations and exposing new behaviors are shown in Figure 6b and Figure 6c, respectively. On average, DueForce outperforms PMP and BDA* by 169.82% and 443.24% on revealing new define-use relations, and by 167.50% and 174.76% on exposing new behaviors. The result also shows that, the more previously uncovered define-use relations are revealed, the more previously hidden syscall sequences are triggered. It demonstrates that compared with branch coverage, define-use coverage is a better proxy for behavior coverage.

We also note that, we run DueForce for an equivalent number of iterations as PMP completes its linear search. It is possible that certain dependencies identified by PMP or BDA* are located within paths that have lower weights in terms of exposing more uncovered define-use relations, and thus have not been identified by DueForce within a limited number of iterations.

Case study. We use a trojan malware sample [48] as a case study. Figure 8 simulates its behaviors. It connects to each computer within the intranet (lines 12-15), brute-forces the username and password combinations (lines 18-21), and sends the correct ones to the attacker's server (lines 24-25). The malicious behaviors need to be executed sequentially, governed by a switch-case within a loop. In a plain execution, the malware terminates rapidly due to the unavailability of the attacker's server. Linear search lacks adequate guidance on generating a path scheme within a loop. It blindly tries to traverse uncovered control flow edges. When no uncovered edge remains within a loop, it allows the loop to continue as usual until the loop count reaches a predefined limit. As a comparison, DueForce predicts three define-use relations from previous executions, including $\langle L13, L18, \texttt{\&usr}\rangle$, $\langle L14, L19, \texttt{\&pwd}\rangle$, and $\langle L12, L24, \texttt{\&S.IP}\rangle$. The predictions guide the path exploration within the loop to sequentially traverse case 0, case 1, and case 2 in an execution. Consequently, the malicious behaviors are exposed.

5 RELATED WORK

Data-flow aware fuzzing. The conventional approach to fuzz testing mostly relies on popular forms of edge coverage [27, 43, 53].Recent advancements have introduced a variety of feedback mechanisms, with data-flow relations emerging as a promising candidate. Such as DDFuze [36] and DAFL [34], both utilizing data dependency relations as feedback mechanisms. Different from using data-flow relations to guide the mutation of test cases, DueForce instead employs data-flow relations to mutate path schemes.

Forced execution. Forced execution techniques, achieved by forcefully setting certain branch outcomes, can expose hidden software behaviors and is widely used in the analysis of malware [40, 49, 52]. As discussed in the introduction section, considerable efforts have been dedicated to crash-free execution and for different platforms [13, 32, 33, 47, 50]. However, limited attention has been devoted to the critical aspect of path exploration. Existing approaches predominantly rely on a linear search algorithm [40, 52], which may traverse a substantial number of unnecessary paths, leaving some paths unexplored that are pivotal for revealing malware behaviors.

Path exploration. The exploration of program execution paths is typically conducted through many methods such as symbolic execution and fuzzing tests. Symbolic execution involves the use of symbolic variables to represent concrete input values, allowing path exploration to cover multiple possible input scenarios while maintaining abstraction [23, 29, 38, 44]. Fuzzing tests generate diverse test inputs to explore various execution paths of the target program [43, 53]. Both of them involve generating inputs capable of traversing a specific path. Whereas in forced execution, path exploration [20, 22, 24, 35, 37, 40, 41, 45, 46, 51, 52] simply requires forcibly setting the outcomes of certain branches without generating the corresponding inputs.

Memory dependence analysis. Since Debray et al. [26] and Cifuentes et al. [25] pioneered the field of memory dependencies analysis for executable files by propagating abstract domains along the registers of each instruction, numerous studies have emerged to address this issue [15, 16, 18, 19, 21, 30, 31, 39, 42, 54]. VSA [18] enhances the ideas of Debray et al. by tracking the value flow along registers and memory locations. DeepVSA [31] further improves VSA by employing a neural network to predict the memory-access regions of each instruction, and subsequently enhancing VSA. BDA [54], to mitigate accuracy loss caused by path merging, utilizes probabilistic analysis for uniformly sampling paths and abstract interpretation. These studies employ conservative analysis to detect memory dependence which leads to over-approximation. DueForce dynamically predicts memory dependency and corrects previous predictions.

6 CONCLUSION

We propose a novel and practical path exploration strategy (called DueForce) for forced execution that features using define-use relations as guidance in a progressive and self-supervised fashion. Compared to prioritizing the coverage of control flow graph edge, following the define-use guidance enables precise path plans for exploring new program behaviors. Consequently, DueForce effectively reduces unnecessary path exploration, incomplete behavior capture, and uninitialized pointer dereferences. We develop a prototype system and apply it to the SPEC2000 benchmark and 200 real-world malware samples. The evaluation results show that DueForce is substantially more effective and efficient than state-of-the-art forced execution tools.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their constructive comments. The RUC authors were supported by the National Natural Science Foundation of China under grants 62002361, 62272465, U1836209 and 62272464, and the Fundamental Research Funds for the Central Universities and the Research Funds of Renmin University of China under grant 22XNKJ29. The Purdue authors were supported by ONR N000142012733, ONR N000142312081, NSF-CCF1910300 and NSF-CCF1901242. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] 2018. API hooks of Cuckoo. https://github.com/cuckoosandbox/monitor/tree/master/sigs.

[2] 2018. Evil Clone Attack. https://gbhackers.com/evil-clone-attack-legitimate-pdf-software.

[3] 2018. Padawan. https://padawan.s3.eurecom.fr/about.

[4] 2019. Fileless Malware. https://www.cybereason.com/blog/fileless-malware.

[5] 2020. Cuckoo. https://cuckoosandbox.org/.

[6] 2020. Cybercrime To Cost The World $10.5 Trillion Annually By 2025. https://cybersecurityventures.com/cybercrime-damages-6-trillion-by-2021/.

[7] 2023. VirusShare. https://virusshare.com/.

[8] 2024. Clickless PowerPoint Malware Installs When Users Hover Over a Link. https://blog.barkly.com/powerpoint-malware-installs-when-users-hover-over-a-link.

[9] 2024. Details of Malware Analysis Result. https://www.virustotal.com/gui/file/3110f00c1c48bbba24931042657a21c55e9a07d2ef315c2eae0a422234623194.

[10] 2024. Malware Anafor Case Study. https://github.com/DueForce/DueForce/blob/main/MalwareAnalysis.pdf.

[11] 2024. Pin - A Dynamic Binary Instrumentation Tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[12] 2024. Qemu User Emulation. https://wiki.debian.org/QemuUserEmulation.

[13] A. Abraham, R. Andriatsimandefitra, A. Brunelat, J.-F. Lalande, and V. Viet Triem Tong. 2015. GroddDroid: a gorilla for triggering malicious behaviors. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. 119–127. https://doi.org/10.1109/MALWARE.2015.7413692

[14] Australian Digital Health Agency. 2022. Cyber Security Report 2022. https://www.digitalhealth.gov.au/sites/default/files/documents/cyber-security-report-2022.pdf.

[15] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. 2013. Production-Run Software Failure Diagnosis via Hardware Performance Counters. *SIGARCH Comput. Archit. News* 41, 1 (mar 2013), 101–112. https://doi.org/10.1145/2490301.2451128

[16] Joy Arulraj, Guoliang Jin, and Shan Lu. 2014. Leveraging the Short-Term Memory of Hardware to Diagnose Production-Run Software Failures. *SIGPLAN Not.* 49, 4 (feb 2014), 207–222. https://doi.org/10.1145/2644865.2541973

[17] ATA. 2023. SPEC2000. http://www.spec2000.com/.

[18] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*. Springer, 5–23.

[19] Gogul Balakrishnan and Thomas Reps. 2010. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 32, 6 (2010), 1–84.

[20] Leyla Bilge, Davide Balzarotti, William Robertson, Engin Kirda, and Christopher Kruegel. 2012. Disclosure: detecting botnet command and control servers through large-scale netflow analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM.

[21] David Brumley and James Newsome. 2006. *Alias analysis for assembly*. Technical Report. Technical Report CMU-CS-06-180, Carnegie Mellon University School of ....

[22] Ahmet Salih Buyukkayhan, Alina Oprea, Zhou Li, and William Robertson. 2017. Lens on the endpoint: Hunting for malicious software through endpoint data analysis. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer.

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

[24] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. *SIGPLAN Not.* 47, 4 (March 2011). https://doi.org/10.1145/2248487.1950396

[25] Cristina Cifuentes and Antoine Fraboulet. 1997. Intraprocedural static slicing of binary executables. In *1997 Proceedings International Conference on Software Maintenance*. IEEE, 188–195.

[26] Saumya Debray, Robert Muth, and Matthew Weippert. 1998. Alias analysis of executable code. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 12–24.

[27] M. Eddington. 2014. "Peach fuzzing platform". http://community.peachfuzzer.com/WhatIsPeach.html.

[28] ESET. 2024. ESET Anti-Virus Software. https://www.eset.com/.

[29] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 213–223.

[30] Bolei Guo, Matthew J Bridges, Spyridon Triantafyllis, Guilherme Ottoni, Easwaran Raman, and David I August. 2005. Practical and accurate low-level pointer analysis. In *International Symposium on Code Generation and Optimization*. IEEE, 291–302.

[31] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. 2019. {DEEPVSA}: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis. In *28th USENIX Security Symposium (USENIX Security 19)*. 1787–1804.

[32] Xunchao Hu, Yao Cheng, Yue Duan, Andrew Henderson, and Heng Yin. 2018. Jsforce: A forced execution engine for malicious javascript detection. In *Security and Privacy in Communication Networks: 13th International Conference, SecureComm 2017, Niagara Falls, ON, Canada, October 22–25, 2017, Proceedings 13*. Springer, 704–720.

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

[34] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. {DAFL}: Directed Grey-box Fuzzing guided by Data Dependency. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4931–4948.

[35] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and Xiaofeng Wang. 2009. Effective and efficient malware detection at the end host. In *USENIX 2009, 18th Usenix Security Symposium*. http://www.eurecom.fr/publication/2774

[36] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. 2022. Fuzzing with data dependency information. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 286–302.

[37] Tipp Moseley, Dirk Grunwald, Daniel A Connors, Ram Ramanujam, Vasanth Tovinkere, and Ramesh Peri. 2006. Loopprof: Dynamic techniques for loop detection and profiling. In *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA)*.

[38] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Exploring Multiple Execution Paths for Malware Analysis. In *2007 IEEE Symposium on Security and Privacy (SP '07)*. 231–245. https://doi.org/10.1109/SP.2007.17

[39] Kexin Pei, Dongdong She, Michael Wang, Scott Geng, Zhou Xuan, Yaniv David, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2022. NeuDep: neural binary memory dependence analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 747–759.

[40] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-Force: Force-Executing Binary Programs for Security Applications. In *Proceedings of the 23rd USENIX Security Symposium*.

[41] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[42] Thomas Reps and Gogul Balakrishnan. 2008. Improved memory-access analysis for x86 executables. In *International Conference on Compiler Construction*. Springer, 16–35.

[43] R.Swiecki and F.Grobert. 2010. "honggfuzz". https://github.com/google/honggfuzz.

[44] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 263–272.

[45] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS '08)*. Springer-Verlag. https://doi.org/10.1007/978-3-540-89862-7_1

[46] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16.

[47] Zhenhao Tang, Juan Zhai, Minxue Pan, Yousra Aafer, Shiqing Ma, Xiangyu Zhang, and Jianhua Zhao. 2018. Dual-force: Understanding webview malware via cross-language forced execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 714–725.

[48] VirusTotal. 2024. Malware Report for Case Study. https://www.virustotal.com/gui/file/0bd4ab6ed21edca2195159026913ec617adb69b63c3ee2adb37e1932700809be.

[49] Liang Xu, Fangqi Sun, and Zhendong Su. 2010. Constructing Precise Control Flow Graphs from Binaries. https://api.semanticscholar.org/CorpusID:15538708

[50] Lei Xue, Yajin Zhou, Ting Chen, Xiapu Luo, and Guofei Gu. 2017. Malton: Towards On-Device Non-Invasive Mobile Malware Analysis for ART. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 289–306. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/xue

[51] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM. https://doi.org/10.1145/1315245.1315261

[52] Wei You, Zhuo Zhang, Yonghwi Kwon, Yousra Aafer, Fei Peng, Yu Shi, Carson Harmon, and Xiangyu Zhang. 2020. PMP: Cost-effective Forced Execution with Probabilistic Memory Pre-planning. In *2020 IEEE Symposium on Security and*

Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. IEEE, 1121–1138. https://doi.org/10.1109/SP40000.2020.00035
[53] M. Zalewski. 2014. "American Fuzzy Lop". http://lcamtuf.coredump.cx/afl/.
[54] Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. 2019. BDA: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 137:1–137:31.

Received 16-DEC-2023; accepted 2024-03-02