2020 IEEE Symposium on Security and Privacy

PMP: Cost-effective Forced Execution with Probabilistic Memory Pre-planning

Wei You[1], Zhuo Zhang[1], Yonghwi Kwon[2], Yousra Aafer[1], Fei Peng[1], Yu Shi[1], Carson Harmon[1], Xiangyu Zhang[1]
[1]Department of Computer Science, Purdue University, Indiana, USA
[2]Department of Computer Science, University of Virginia, Virginia, USA
Email: {you58, zhan3299, yaafer, pengf, shi442, harmon35, xyzhang}@purdue.edu, yongkwon@virginia.edu

***Abstract*—Malware is a prominent security threat and exposing malware behavior is a critical challenge. Recent malware often has payload that is only released when certain conditions are satisfied. It is hence difficult to fully disclose the payload by simply executing the malware. In addition, malware samples may be equipped with cloaking techniques such as VM detectors that stop execution once detecting that the malware is being monitored. Forced execution is a highly effective method to penetrate malware self-protection and expose hidden behavior, by forcefully setting certain branch outcomes. However, an existing state-of-the-art forced execution technique X-Force is very heavy-weight, requiring tracing individual instructions, reasoning about pointer alias relations on-the-fly, and repairing invalid pointers by on-demand memory allocation. We develop a light-weight and practical forced execution technique. Without losing analysis precision, it avoids tracking individual instructions and on-demand allocation. Under our scheme, a forced execution is very similar to a native one. It features a novel memory pre-planning phase that pre-allocates a large memory buffer, and then initializes the buffer, and variables in the subject binary, with carefully crafted values in a random fashion before the real execution. The pre-planning is designed in such a way that dereferencing an invalid pointer has a very large chance to fall into the pre-allocated region and hence does not cause any exception, and semantically unrelated invalid pointer dereferences highly likely access disjoint (pre-allocated) memory regions, avoiding state corruptions with probabilistic guarantees. Our experiments show that our technique is 84 times faster than X-Force, has 6.5X and 10% fewer false positives and negatives for program dependence detection, respectively, and can expose 98% more malicious behaviors in 400 recent malware samples.**

I. INTRODUCTION

The proliferation of new strains of malware every year poses a prominent security threat. Recently reported attacks demonstrate the emergence of new attacking trends, where malware authors are designing for stealth and leaving lighter footprints. For example, Fileless malware [5] infects a target host through exploiting built-in tools and features, without requiring the installation of malicious programs. Clickless infections [1] avoid end-user interaction through exploiting shared access points and remote execution exploits. Cryptocurrency malware [4] allow attackers to generate huge revenues by illegally running mining algorithms using victim's system resources. According to [3], a massive cryptocurrency mining botnet has generated \$3 million revenue in 2018. Under this new threatscape, malicious payloads have evolved and look much different than traditional ones. Thus, a critical challenge the security community is facing today is to understand and analyze emerging malware's behavior in an effort to prevent potentially epidemic consequences.

A popular approach to understanding malware behavior is to run it in a sandbox. However, a well-known difficulty is that the needed environment or setup may not be present (e.g., C&C server is down and critical libraries are missing) such that the malware cannot be executed. In addition, recent malware often makes use of time-bomb and logic-bomb that define very specific temporal and contextual conditions to release payload, and some samples even use cloaking techniques such as packing, and VM/debugger detectors that prevent execution when the malware is being monitored.

Researchers in [32] proposed a technique called *forced-execution* (X-Force) that penetrates these malware self-protection mechanisms and various trigger conditions. It works by force-setting branch outcomes of some conditional instructions. (e.g., those checking trigger conditions). As forcing execution paths could lead to corrupted states and hence exceptions, X-Force features a *crash-free execution model* that allocates a new memory block on demand upon any invalid pointer dereference. However, X-Force is a very heavy-weight technique that is difficult to deploy in practice. Specifically, in order to respect program semantics, when X-Force fixes an invalid pointer variable (by assigning a newly allocated memory block to the variable), it has to update all the correlated pointer variables (e.g., those have constant offsets with the original invalid pointer). To do so, it has to track all memory operations (to detect invalid accesses) and all move/addition/subtraction operations (to keep track of pointer variable correlations/aliases). Such tracking not only entails substantial overhead, but also is difficult to implement correctly due to the complexity of instruction set and the numerous corner situations that need to be considered (e.g., in computing pointer relations). As a result, the original X-Force does not support tracing into library functions.

In this paper, we propose a practical forced execution technique. It does not require tracking individual memory or arithmetic instructions. Neither does it require on demand memory allocation. As such, the forced execution is very close to a native execution, naturally handling libraries and dynamically generated code. Specifically, it achieves crash-free execution (with probabilistic guarantees) through a novel memory pre-planning phase, in which it pre-allocates a region of memory starting from address 0, and fills the region with carefully crafted random values. These values are designed in such a way that (1) if they are interpreted as addresses and further dereferenced, the addresses fall into the pre-allocated region and do not cause exception; (2) they have diverse

DOI 10.1109/SP40000.2020.00035

random values such that semantically unrelated pointer variables unlikely dereference the same random address and avoid causing bogus program dependencies and corrupted states. An execution engine is developed to systematically explores different paths by force-setting different sets of branch outcomes. For each path, multiple processes are spawned to execute the path with different randomized memory pre-planning schemes, further reducing the probability of coincidental failures. The results of these processes are aggregated to derive the results for the particular path. The engine then moves forward to the next path.

Our contributions are summarized as follows.

- We develop a practical forced-execution engine that does not entail any heavy-weight instrumentation.
- We propose a novel memory pre-planning scheme that provides probabilistic guarantees to avoid crashes and bogus program dependencies. The execution under our scheme is very similar to a native execution. Once the memory is pre-planned and initialized at the beginning, the execution just proceeds as normal, without requiring any tracking or on the fly analysis (e.g., pointer correlation analysis).
- We have implemented a prototype called PMP and evaluated it on SPEC2000 programs (which include `gcc`), and 400 recent real-world malware samples. Our results show that PMP is a highly effective and efficient forced execution technique. Compared to X-Force, PMP is 84 time faster, and the false positive (FP) and false negative (FN) rates are 6.5X and 10% lower, respectively, regarding dependence analysis; and detect 98% more malicious behaviors in malware analysis. It also substantially supersedes recent commercial and academic malware analysis engines Cuckoo [2], Habo [10] and Padawan [8].

II. MOTIVATION

In this section, we use an example to motivate the problem, explain the limitations of existing techniques, and illustrate our idea. The code snippet in Figure 1 simulates the command and control (C&C) behavior of a variant of Mirai [7], a notorious IoT malware that launches distributed denial of service attacks when receiving commands from the remote C&C server. In particular, it reads the maximum number of destination hosts (to attack) from a configuration file (line 9), and allocates a `Cmd` object with sufficient memory to store destination information in the `Dest` objects (lines 10-12). When the C&C server is connectable (line 15), the malware scans the local network for the destination hosts (line 16), receives the requested command (line 17), and performs the corresponding actions on the destination hosts (lines 18-22).

To expose such malicious behavior, analysts could run the sample in a sandbox and monitor its system call sequences and network flows [8]. Unfortunately, a naive execution-based analysis is incomplete and hence cannot reveal all the malicious payloads, especially those that are condition-guarded and environment-specific. In our example, if the configuration file does not exist or the C&C server is not connectable, the malicious behavior will not be exposed at all. One may consider to construct an input file and simulate the network data. However, such a task is time-consuming and not practical for zero-day malware whose input format and network communication protocol are unknown. In addition, recent malware samples are increasingly equipped with anti-analysis mechanism, which prevents these samples from execution even if they are given valid inputs (please refer to Section IV for real-world cases). This poses great difficulties for dynamic analysis.

Forced execution [32] provides a practical solution to systematically explore different execution paths (and, hence reveal different program behaviors) without any input or environment setup. It works by force-setting branch outcomes of a small set of predicates and jump tables. One critical problem faced by forced execution is invalid memory accesses due to the absence of necessary memory allocations and initializations, which are present in normal execution. Without appropriate handling of invalid memory accesses, the program is most likely to crash before reaching any malicious payload. In our example, the malicious behaviors were supposed to be exposed, if the predicate in line 15 is forced to take the `true` branch, and the jump table in line 18 is forced to iterate different entries. However, the forced execution fails in line 30, because `cmd` is not properly allocated and its `dests` field is not initialized.

X-Force. In X-Force [32], researchers show that simply ignoring exceptions does not work as that leads to cascading failures (i.e., more and more crashes), they propose to recover from invalid memory accesses by performing on-demand memory allocation. In particular, X-Force monitors all memory operations (i.e., allocate, free, read and write) to maintain a list of valid memory addresses. If an accessed memory address is not in the valid list, a new memory block will be allocated on demand for the access. To respect program semantics, when a pointer variable holding an invalid address x is set to the address of the allocated memory, all the other pointer variables that hold a value denoting the same invalid address or its offset (e.g., $x + c$ with c some constant) need to be updated. X-Force achieves this through *linear set tracing*, which identifies linearly correlated pointer variables that are induced by address offsetting. When a pointer variable is updated, all the correlated pointers in its linear set need to be updated accordingly based on their offsets.

Assume in an execution instance, line 8 takes the `false` branch and line 15 is forced to take the `true` branch. In this execution, `cmd` is a `NULL` pointer, hence the `dests` pointer in line 27 points to 0x8 (the offset of `dests` field is 8). The rounded rectangle in Figure 1 illustrates what X-Force does for the memory access of `dests[0]->ip` in line 30. Linear sets are maintained for each register and each memory address. In particular, $SR(r)$ and $SM(a)$ are used to denote the linear set of register r and address a, respectively. After executing instruction α, the linear set of register `rbx` is updated to be the same as that of `&dests`, i.e., $SR(\texttt{rbx}) \leftarrow SM(\texttt{\&dests})$ such that $SR(\texttt{rbx})$=$SM(\texttt{\&dests})$=$\{$0x7ffdffffed0$\}$, which

```
typedef struct{char ip[16]; long port;} Dest;
typedef struct{long act; Dest* dests[0];} Cmd;

int main(int argc, char *argv[]) {
  Cmd *cmd = NULL;
  int max = 0;

  if (config_file_exists()) {
    max = read_from_config_file();
    cmd = malloc(sizeof(Cmd) + max*sizeof(Dest*));
    for (int i = 0; i < max; i++)
      cmd->dests[i] = malloc(sizeof(Dest));
  }
  ...
  if (cnc_server_connectable()) {
    scan_intranet_hosts(cmd, max);
    cmd->act = get_action_from_cc_server();
    switch (cmd->act) {
      case 1: do_action_1(cmd->dest, max); break;
      case 2: do_action_2(cmd->dest, max); break;
      ...
    }
  }
  ...
}
```

```
void scan_intranet_hosts(Cmd *cmd, int max) {
  Dest **dests = cmd->dests;
  for (int i = 0; i < max; i++) {
    struct sockaddr_in *host = iterate_host();
    inet_ntop(host->ip, dests[i]->ip);
    dests[i]->port = ntohl(host->port);
  }
}
```

```
α. mov rbx, [rbp - 0x10]   // rbx = [rbp - 0x10] = [0x7ffdfffffed0] = 0x8
    /* Validate Memory Address: get_accessible(0x7ffdfffffed0) = true */
    /* Update Linear Set: SR(rbx) ← SM(&dests) = {0x7ffdfffffed0} */
β. mov ecx, [rbp - 0x14]   // ecx = [rbp - 0x14] = [0x7ffdfffffecc] = 0x0
    /* Validate Memory Address: get_accessible(0x7ffdfffffed4) = true */
    /* Update Linear Set: SR(rcx) ← SM(&i) = {0x7ffdfffffecc} */
γ. lea rdx, [rbx + 8*rcx]  // rdx = rbx + 8*rcx = 0x8
    /* Update Linear Set: SR(rdx) ← SR(rbx) = {0x7ffdfffffed0} */
δ. mov rax, [rdx]  // rax = [rdx] = [0x8]
    /* Validate Memory Address: get_accessible(0x8) = false (invalid read on 0x8) */
    /* Allocate Memory Block: malloc(BLOCK_SIZE) = 0x2531000 */
    /* Update Reference: rdx = *(0x7ffdfffffed0) = 0x2531000 + 0x8 = 0x2531008 */
ϵ. mov rax, [rax]  // rax = [rax] = [0x0]
    /* Validate Memory Address: get_accessible(0x0) = false (invalid read on 0x0) */
    /* Allocate Memory Block: malloc(BLOCK_SIZE) = 0x2532000 */
    /* Update Reference: rdx = *(0x7ffdfffffed0) = 0x2532000 + 0x8 = 0x2532008 */
```

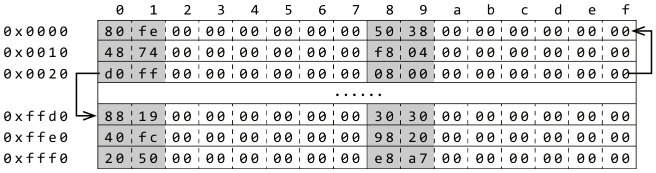
Fig. 1: Motivation example. The assembly code here is functionally equivalent with the original one for easy understanding.

is the address of `dests`. Intuitively, the pointer value in rbx is linearly correlated to that in `dests`. Hence, fixing either one entails updating the other. The linear correlation is further propagated to register `rdx` after executing instruction γ, since its value is derived from `rbx` by address offsetting (i.e., `&dests[0]` = `&dests` + 0). When executing instruction δ, X-Force detects an invalid access through the pointer denoted by `rdx` (i.e., `&dests[0]`), holding an invalid address 0x8. Hence, it allocates a memory block with address 0x2531000 and initializes it with zero values. Register `rdx` is then updated to 0x2531008. The value of `&dest` should also be updated, since it linearly correlates with `rdx`. Similar memory recovery operations are needed for instruction ϵ that accesses `dests[0]->ip` through an invalid memory address 0x0.

As we can see that each memory operation should be intercepted by X-Force for memory address validation and linear set tracing. Upon the recovery of an (invalid) pointer variable, all the linearly correlated variables need to be updated accordingly. This causes substantial performance degradation. It was reported that X-Force has 473 times runtime overhead over the native execution [32]. Furthermore, since many library functions such as string functions in `glibc` can lead to linear set explosion (due to substantial heap array operations), X-Force chose not to trace into library functions to update linear sets. As a result, its memory recovery is incomplete (see Section IV for a real-world example).

Our technique. We propose a novel randomized memory pre-planning technique (called PMP) to handle invalid memory accesses with probabilistic guarantees. Instead of allocating new memory blocks on demand, PMP pre-allocates a large memory block with a fixed size (e.g., 16KB) when the program is loaded. The *pre-allocated memory area* (PAMA) is filled with carefully crafted random values such that if these values are interpreted as memory addresses, the corresponding accesses still fall into PAMA. We call this *self-contained memory behavior* (SCMB). In addition, these random values are designed in a way that they are self-disambiguated. That is, it is highly unlikely that two semantically unrelated memory operations access the same random address, causing bogus dependencies. We call this *self-disambiguated memory behavior* (SDMB). For example, the simplest way to achieve SCMB is to pre-allocate a chunk of memory starting at 0x00 and fill it with 0x00. As such, dereferences of null pointers (e.g., $*p$ with $p = 0$) or pointers with some offset from null (e.g., $*(p+8)$), yield value 0x00 due to the initialization. If the yielded value 0x00 is further interpreted as a pointer, its dereference continues to yield 0x00, without causing any memory exception. However, such a scheme leads to substantial bogus program dependencies as semantically unrelated memory operations through uninitialized/invalid pointer variables all end up accessing address 0x00. For example, assume p and q are not properly initialized and both have a null value due to forced execution and there are two pointer dereference statements "1. $*\,p$ = ...;   2. ... = $*q$". A bogus dependence will be introduced between 1 and 2. Such bogus dependencies further lead to highly corrupted program states. SDMB is to ensure that unrelated pointer variables have a high likelihood to contain disjoint addresses such that it is like they were all properly allocated and initialized. Intuitively, PMP diversifies the values filled in the pre-allocated large memory region such that dereferences at different offsets yield different values. Consequently, follow-up dereferences (of these values) can continue to disambiguate themselves.

In addition to the aforementioned pre-planning, during execution, PMP also initializes global, local variables, and heap regions *allocated by the original program logic* with random values pointing to PAMA. Note that otherwise they are initialized to 0 by default. As such, when these variables are interpreted as pointers and dereferenced without being



```
        0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
0x0000  80 fe 00 00 00 00 00 00 50 38 00 00 00 00 00 00
0x0010  48 74 00 00 00 00 00 00 f8 04 00 00 00 00 00 00
0x0020  d0 ff 00 00 00 00 00 00 08 00 00 00 00 00 00 00
                            ......
0xffd0  88 19 00 00 00 00 00 00 30 30 00 00 00 00 00 00
0xffe0  40 fc 00 00 00 00 00 00 98 20 00 00 00 00 00 00
0xfff0  20 50 00 00 00 00 00 00 e8 a7 00 00 00 00 00 00
```

Fig. 2: Pre-allocated memory area. The data is presented in the little-endian format for the x86_64 architecture. The bytes in gray are free to be filled with 8-multiple random values.

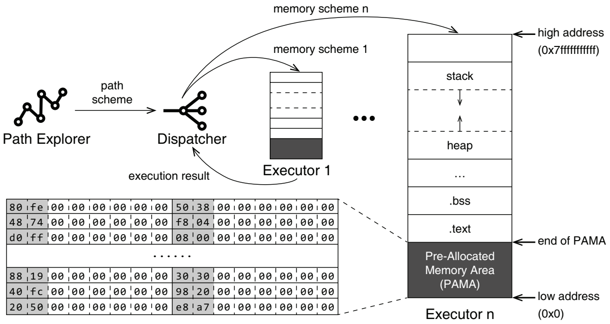


Fig. 3: Architecture of PMP.

properly initialized along some forced path, the accesses still fall in PAMA and also have low likelihood to collide (on the same address). Through SCMB, PMP enables crash-free memory operations, which are critical for forced execution. Since it does not require tracing memory operations or performing on-demand allocation, it is 84 times faster than X-Force (Section IV). Through SDMB, PMP respects program semantics such that it can faithfully expose (hidden) program behaviors with probabilistic guarantees. As shown in our evaluation (Section IV), PMP has fewer false positives (FP) and false negatives (FN) than X-Force as well.

Figure 2 illustrates a 64-KB pre-allocated memory area mapped in the address space from 0x0 to 0xffff. Note that although this memory region may overlap with some reserved address ranges, we leverage QEMU's address mapping to avoid such overlap (see Section III-E). It is filled with crafted random values that ensure both SCMB and SDMB. For our motivation example, instruction δ reads the memory unit at address 0x8 (i.e., `&dests[0]`) and gets the value 0x3850. Subsequently, the instruction ϵ uses 0x3850 as the address to access `dests[0]->ip`. These two accessed addresses (0x8, 0x3850) are contained in the PAMA, hence no memory exception occurs. The data dependence between these two addresses are also faithfully exposed, without undesirable address collision. Observe that there is no memory validation and linear set tracing required.

We want to point out while SCMB and SDMB can be effectively ensured in forced execution, they may not be as effective in regular execution. Otherwise, dynamic memory allocation could be completely avoided. The reason is that forced execution aims to achieve good coverage to expose program behaviors such that it bounds loop iterations [32]. As a result, linear scannings of large memory regions are mostly avoided, allowing to establish SCMB and SDMB effectively and efficiently. Intuitively, one can consider that our design is equivalent to pre-allocating many small regions that are randomly distributed. This is particularly suitable for heap accesses in forced-execution as they tend to happen in smaller memory regions. Even if overflows might happen, the likelihood of critical data being over-written is low due to the random distribution.

III. DESIGN

A. Overview

Figure 3 presents the architecture of PMP, which consists of three components: the path explorer, the dispatcher and the executors. Given a target binary, the path explorer systematically generates a sequence of branch outcomes to enforce, including the PCs of the conditional instructions and their true/false values. We call it a *path scheme*. Note that like X-Force, PMP does not enforce the branch outcome of all predicates, but rather just a very small number of them (e.g., less than 20). The other predicates will be evaluated as usual. PMP operates in rounds, each round executing a path scheme. For each path scheme, PMP further generates multiple versions of variable initializations, each having different initial values but satisfying both SCMB and SDMB. We call them *memory schemes*. The reason of having multiple memory schemes is to reduce the likelihood of coincidental address collisions. A process is forked for each path and memory scheme and distributed to an executor for execution. At the end of a round, the dispatcher aggregates the results from the executors (e.g., coverage). Another path scheme is then computed by the path explorer to get into the next round, based on the results from previous rounds.

Path Explorer. In essence, path exploration is a search process that aims to cover different parts of the subject binary. In each round, a new path scheme is determined by switching additional/different predicates, or enforcing additional/different jump table entries, to improve code coverage. Since the search space of all possible paths is prohibitively large for real-world binaries, PMP follows the same path exploration strategies in X-Force [32], including the linear search, the quadratic search and the exponential search. In particular in each round, the linear search selects a new predicate or jump table entry to enforce, which is usually the last one that does not have all its branches covered in previous rounds. The exponential strategy aims to explore all combinations of branch outcomes and is hence the most expensive. It is only used to explore some critical code regions. Quadratic search falls in between the two. Since these are not our contributions, interested readers are referred to the X-Force project [32].

Dispatcher. The dispatcher aggregates execution results (e.g., code coverage and program dependencies) of multiple executors in a conservative fashion. Specifically, it considers a result valid if and only if it is agreed by n executors, with n configurable. In our experience, $n = 2$ is good enough in practice. Such aggregation further improves our

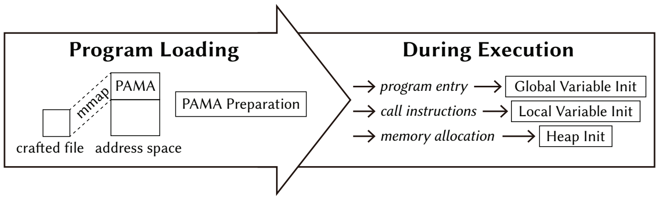


Fig. 4: Workflow of Memory-preplanning.

probabilistic guarantees. Intuitively, assume PMP ensures that a reported result has lower than $p \in [0, 1]$ probability to be incorrect during a single execution (on an executor), due to the inevitable accidental violations of SCMB or SDMB. The aggregation further reduces the probability to p^n if the memory schemes on the various executors are truly randomized (and hence independent).

Executors. All executors are forked from the same main process with the same initialized PAMA. Each executor then enforces a given path and memory scheme assigned to it. Such a design avoids the redundant initialization of PAMA. Note that all memory accesses must start from some variable, whose value is fully randomized across executors.

The rest of this section will explain in details the memory pre-planning step and the probability analysis for SCMB and SDMB guarantees. Execution result aggregation is omitted due to its simplicity.

B. Memory Pre-planning

Overview. Figure 4 presents the workflow of memory pre-planning. When a program is loaded, a pre-allocated memory area (PAMA) is prepared by invoking the `mmap` system call to map a crafted file to the program address space. The file content is randomly generated beforehand. During execution, program variables (including global, local variables and heap regions) are initialized by PMP with random eight-multiple values pointing to PAMA. Specifically, PMP intercepts: 1) the program entry point for initializing global variables; 2) call instructions for initializing local variables; and 3) memory allocations for initializing heap regions. Note that PAMA preparation happens a priori and incurs negligible runtime overhead, while variable initialization occurs on-the-fly during execution. Both are generic and do not require case-by-case crafting. We further discuss these steps in the following.

PAMA Preparation. PAMA is mapped at the lower part of the address space starting from 0x0, in order to accommodate null pointers or pointers with invalid small values. The word-aligned addresses within PAMA (i.e., those having 0 at the lowest three bits) are filled with carefully crafted random values, such that if these values are interpreted as addresses, they fall within PAMA. As such, the range of random values that we can fill is dependent on the size of PAMA. For a 64-KB PAMA (i.e., in the address range of [0, 0xffff]), the first two least-significant bytes of a filling value are free to be set with a random eight-multiple value. Other bytes are fixed to zero. Note that such a value is essentially a valid word-aligned address in PAMA. For a 64-MB PAMA, the first three least-significant bytes of a filling value can be set randomly, providing better SDMB. The maximum PAMA can be as large as 128 TB, as a larger PAMA would overlap with the kernel space. While a feasible design is to change the entire virtual space layout (by changing kernel), it would hinder the applicability of our technique. In practice, we find that 4-MB of PAMA provides a good balance of SCMB and SDMB.

Global Variable Initialization. In an ELF binary, the uninitialized or zero-initialized global variables are stored in the `.bss` segment. During loading, PMP reads the offset and size information of the `.bss` segment from the ELF header. PMP then initializes the segment like a heap region.

Heap Initialization. Pre-planning heap regions that are dynamically allocated by instructions in the subject binary is relatively easier. PMP intercepts all memory allocations and set the allocated regions to contain random word-aligned PAMA addresses. Note that PMP writes these values to each word-aligned address in the heap region. If a regular compiler is used to generate the subject binary, the compiler would enforce pointer-related memory accesses to be word-aligned through padding. However, malware may intentionally introduce pointer accesses that are not word-aligned. Section III-E will discuss how PMP handles such cases. In the following discussion, we always assume word alignment.

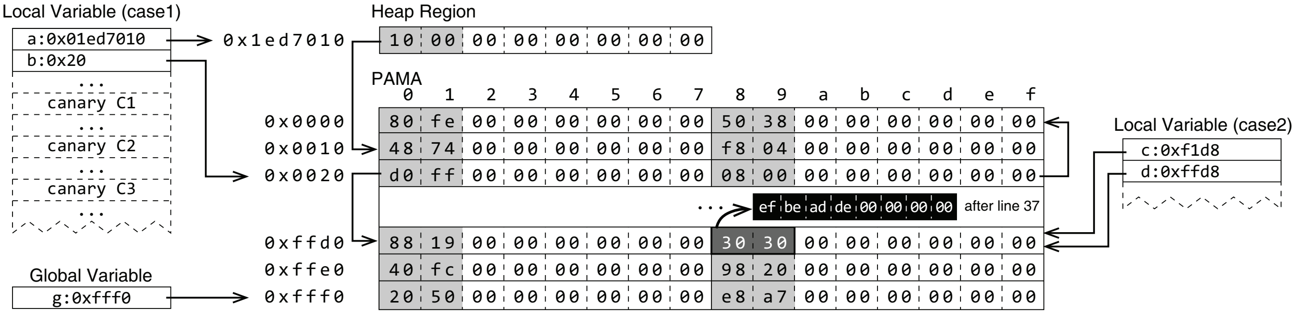
Local Variable Initialization. Initializing local variables is more complex. After initializing PAMA and before spawning the executors, PMP initializes the entire stack region like a heap region. Note that stack frames are pushed and popped frequently and the same stack address space may be used by many function calls. As such, the stack space may need to be re-initialized. A plausible solution is to identify stack frame allocations (e.g., updates of rsp register) and conduct initialization after each allocation. However, due to the flexibility of stack allocations, it is difficult to precisely identify them. Inspired by stack canaries used to detect stack overflows, PMP uses the following design to initialize stack regions. It intercepts each function invocation. Then starting from the current address denoted by rsp, it randomly checks eight [1] unevenly distributed addresses lower than the rsp address (i.e., the potential stack space to be allocated), in the order from high to low, to see if they are PAMA addresses (meaning that they were not overwritten by previous function invocations). We also call these addresses *canaries* without causing confusion in our context and use C_i to denote the ith canary. PMP identifies the lowest (last) canary that is not PAMA address, say C_t, and then re-initializes [C_{t+1}, rsp] (note that stack grows from high address to low address). If all eight canaries are overwritten, PMP continues to check the next eight. Observe that since stack writes may not be continuous, the detection scheme has only probabilistic guarantees. In practice, our scheme is highly

---

[1] Eight is an empirical choice and works well in our evaluation. The number and the distribution of canaries are configurable.

```
01 typedef struct{double *f1; long *f2;} T;
02 typedef struct{char f3; long *f4; long *f5;} G;
03 G *g;
04
05 void case3() {
06   long *e = NULL, *f = NULL;
07   if (cond1()) init(e, f);
08   if (cond2()) {
09     *e = 0x6038; // [0x0000] = 0x6038
10     long tmp = *f; // tmp = [0x0000]: bogus dep!
11   }
12 }
13
14 void case4() {
15   if (cond1()) init(g);
16   if (cond2()) {
17     *(g->f4) = 0x0830;
18     long tmp = *(g->f5); // &(g->f5) = 0x10000
19   }
20 }
21 void case1() {
22   long **a = malloc(...);
23   T *b;
24   if (cond1()) init(b);
25   if (cond2()) {
26     long *alias = b->f2;
27     *(b->f2) = **a; // [0x0008] = [0x0010]
28     *(b->f1) = 0.1; // [0xffd0] = 0.1
29     long tmp = *alias;
30   }
31 }
32
33 void case2() {
34   long *c; double **d;
35   if (cond1()) init(c, d);
36   if (cond2()) {
37     *c = 0xdeadbeef;  // [0xffd8] = 0xdeadbeef
38     double tmp = **d; // [0xdeadbeef]: error!
39   }
40 }
```

(a) code snippet.



(b) memory scheme.

Fig. 5: Memory pre-planning.

effective and we haven't encountered any problems caused by incorrect stack initialization.

Example. We use the code snippet shown in Figure 5a as an example to explain the memory pre-planning process. In the code, a global variable `g` is defined at line 3, two local variables `a`, `b` are defined in function `case1()`. Assume in an execution instance, line 24 takes the `false` branch and `b` is not allocated and initialized; and line 25 is forced to take the `true` branch. Although `a` is initialized by the original program code with an allocated heap region, the data in the heap region is not initialized. Without memory pre-planning, the program would have exception at any of the memory operations in lines 26-29.

In this example, the global variable `g` is set to a random PAMA address at the beginning. Upon calling `case1()`, PMP checks the canaries at C_1, C_2, and so on (see the stack frame in the top-left corner of Figure 5b), and then identifies, say, the region from [C_3,rsp] needs re-initialization, which includes local variables `a` and `b`. Inside the function body, `a` is set to a dynamically allocated heap region at line 22, but other variables such as `g` and `b` keep their initial PAMA address value (as line 24 is not executed). Specifically, `g` and `b` point to 0xfff0 and 0x20 (in PAMA), respectively. Consider the read operation at line 28 that triggers pointer dereferences on `b` and then `b->f1`. The former dereferences address 0x20 and yields value 0xffd0, which is further interpreted as an address in the follow-up dereference of `b->f1`, yielding another valid PAMA address. Observe that any following dereferences will be within PAMA and do not cause any exceptions, illustrating the SCMB property. The value of `b->f1` (i.e., 0xffd0) dereferenced at line 28 is different from that of `b->f2` (i.e. 0x08) dereferenced at line 27, and hence disambiguate themselves, illustrating SDMB.

C. Other PAMA Memory Behavior and Interference with Regular Memory Operations.

Memory pre-planning is particularly designed to handle exceptional memory operations (caused by forced execution). As such, all the values filled in PAMA are essentially in preparation for these values being interpreted as addresses and further dereferenced. It is completely possible that the subject binary does not interpret values from PAMA as addresses. For example, it may interpret a PAMA region as a string and access individual bytes in the region. In such cases, the accessed values are just random values. This is equivalent to how X-Force handles uninitialized/undefined buffers.

A PAMA location can be written to and later read from by instructions in the subject binary, dictated by the program semantics. Program dependencies induced by PAMA are no different from those induced through regular memory regions. For example, the code at line 26 in Figure 5a establishes an alias between variable `alias` and `b->f2`. At line 27, a memory write is conducted on `b->f2`. At line 29, a memory-read is conducted on `alias`. PMP can correctly establish the dependence between line 27 and line 29, since they both point to the same memory address 0x8.

It may happen that a PAMA location is written to by the subject binary and then read through a semantically unrelated invalid pointer dereference later. As the written value may not be a legitimate PAMA address, the later read causes exception. For example, line 37 at function `case2()` of Figure 5a writes a value 0xdeadbeef that is not a word-aligned address within PAMA to the address indicated by pointer `c`. Assume `c` happens to have the same value 0xffd8 as an unrelated pointer `d`. The write to `*c` also changes the value in `*d` to 0xdeadbeef. As such at line 38, an exception is triggered for the read of `**d`. In the next subsection, our probability analysis shows that such cases rarely happen as the likelihood for two semantically unrelated pointers are initialized to the same random value is very low. Furthermore, PMP employs different memory schemes in multiple executors, further reducing such possibility.

In the worst situation, the subject binary uses its own instructions to set semantically unrelated pointers to null. In normal execution, these pointers would point to different properly allocated memory regions. However in forced execution, they may not be allocated, and all point to address 0. In such cases, PMP cannot disambiguate the accesses of these variables, and lead to bogus dependencies. For example, the local variables `e` and `f` in function `case3 ()` of Figure 5a are explicitly set to null by the original program code. In forced execution where line 7 is not executed, they point to the same address 0x0, resulting in bogus dependence (e.g., between lines 9 and 10). Our experimental results in Section IV show that such cases rarely happen.

D. Probability Analysis

In this section, we study the probabilistic guarantee of PMP for the SCMB and SDMB properties. Violations of SCMB lead to exceptions whereas violations of SDMB lead to bogus dependences and corrupted variable values. To facilitate discussion, we introduce the following definitions. Let `PA` be the set of all possible addresses within PAMA, and `WA` be its word-aligned subset. Assume the size of PAMA is S. Then, on a 64-bit architecture, we have equation (1).

$$S = |\mathtt{PA}| = |\mathtt{WA}| \times 8 \tag{1}$$

In addition, let `FV` be a random subset of `WA`, called the *filling value set*, whose elements are used as the values to be filled in PAMA. Without loss of generality, we assume 0 belongs to `FV`. We define the ratio between the size of `FV` and the size of `WA` as *diversity*, denoted as d. Then, we have equation (2).

$$|\mathtt{FV}| = |\mathtt{WA}| \times d = \frac{d \cdot S}{8} \tag{2}$$

The initialization of PAMA can be formulated as a mapping $f : \mathtt{WA} \mapsto \mathtt{FV}$, which assigns each word (with 8 bytes alignment) in PAMA (i.e., denoted by addresses in `WA`) with a random value selected from `FV`. Intuitively, a more diverse `FV` leads to a more random memory scheme. The initialization that fills the whole PAMA with value 0 can be considered an extremal case where `FV` contains only a single element 0. Note that in this case, SCMB is fully respected, while SDMB is substantially violated as all invalid memory operations collide on address 0.

Probabilistic Guarantee of SCMB. When a pointer variable is initialized (by PMP) with a value indicating an address close to the end of PAMA, dereference of its offset may result in an access out of the bound of PAMA. As an example, consider the dereference of `g->f5` at line 18 of function `case4()` in Figure 5a. Recall that `g` is set to be 0xfff0 by PMP. The address of `g->f5` is hence 0x10000, out of the bound of PAMA with 16 KB size.

Theorem 1. Let x be a filling value selected from `FV`, α be an offset. The probability P_{err1} of $x + \alpha$ being out of the bound of PAMA is calculated by equation (3).

$$P_{err1} = P\left((x+\alpha) \notin \mathtt{PA} \mid x \in \mathtt{FV}\right) = \frac{\alpha}{S-8} \cdot \left(1 - \frac{8}{d \cdot S}\right) \tag{3}$$

Proof. For PMP to access an out-of-bound address $x + \alpha$, x must belong to an address set $\mathtt{IA} = \mathtt{WA} \cap \{S-\alpha, S-\alpha+1, \ldots, S-1\}$. To simplify discussion, let $\alpha' = |\mathtt{IA}| = \alpha/8$, $S' = |\mathtt{WA}|$ and $N = |\mathtt{FV}|$. Let the size of $\mathtt{IA} \cap \mathtt{FV}$ be i. We can infer conditional probability $P(x \in \mathtt{IA} \mid x \in \mathtt{FV}) = i/N$, denoted as P_{i1}. Additionally, because there are $\binom{S'-1}{N-1}$ possible `FV`s that could be uniformly chosen from (recall $0 \in \mathtt{FV}$ always holds) and $\binom{\alpha'}{i} \cdot \binom{S'-\alpha'-1}{N-i-1}$ `FV`s have i common elements with `IA`, $P(|\mathtt{FV} \cap \mathtt{IA}| = i) = \binom{\alpha'}{i} \cdot \binom{S'-\alpha'-1}{N-i-1} \Big/ \binom{S'-1}{N-1}$, denoted as P_{i2}. Enumerating size $i \in \{1, \ldots, \alpha'\}$, $P_{err1} = \sum_{i=1}^{\alpha'} P_{i1} \cdot P_{i2} = (\alpha'/N) \cdot \left(\binom{S'-2}{N-2} \Big/ \binom{S'-1}{N-1} \right) = \frac{\alpha}{S-8} \cdot \left(1 - \frac{8}{d \cdot S}\right)$ ☐

Intuitively, the larger the pre-allocated memory area (i.e., S) and the lower the diversity (i.e., d), the lower the P_{err1}. In particular, the P_{err1} of a naive initialization that fills PAMA with value 0 is 0. In a typical setting of $S = 0\text{x}400000$, $\alpha = 8$ and $d = 1$, $P_{err1} = 1.9073\text{e}{-06}$, illustrating a very low chance of exception. A plausible way to completely avoid SCMB violation is to avoid using address values close to the end of PAMA. However this requires knowing the largest possible offset, which is difficult in practice.

Probabilistic Guarantee of SDMB. SDMB will be compromised when two unrelated pointers are initialized to the same value by chance. Taking local variables `c` and `d` for `case2()` in Figure 5a as an example, both of them are initialized to 0xffd8, causing invalid pointer dereference at line 38.

Theorem 2. Let x and y be two filling values independently selected from `FV`. The probability P_{err2} of *coincidental address collision*, when x and y have the same value, is calculated by equation (4).

$$P_{err2} = P\left(x = y \mid x \in \texttt{FV},\ y \in \texttt{FV}\right) = \frac{8}{d \cdot S} \tag{4}$$

Proof. Recall x and y are independently selected from $\texttt{FV}$. Thus, fixing $x = v_0$ as a constant, we can infer $P_{err2} = P(y = v_0 \ |y \in \texttt{FV}) = 1/|\texttt{FV}| = 8/(d \cdot S)$. $\square$

With a typical setting $d = 1$ and $S = 0\text{x}400000$, $P_{err2} = 1.9073\text{e}-06$, a very low probability.

$$\begin{aligned} P_{err3} = & P\left(l\left(x, \beta\right) \cap l\left(y, \gamma\right) \neq \emptyset \mid x \in \texttt{FV},\ y \in \texttt{FV}\right) \\ \leq & \frac{64}{d^2 \cdot S^2} + \left(1 - \frac{8}{d \cdot S}\right)^2 \cdot \frac{\beta + \gamma - 8}{S - 8} \end{aligned} \tag{5}$$

Proof is elided due to space limitations. With a setting of $\beta = 0\text{x}1000$, $\gamma = 0\text{x}1000$, and the rest as the same before, $P_{err3} = 0.00195$, still reasonably low. Note that one can always improve the guarantee by having more executors with different pre-plans.

E. Implementation

PMP is implemented based on the QEMU user-mode emulator [9]. Specifically, PMP instruments conditional jumps and indirect jumps to enforce path scheme. A path scheme is a sequence of branch outcomes that need to be enforced. As an instance, "401a4c:T, 4094fc:F, 40a322#40a566" is a path scheme that contains three branch outcomes to be enforced in order. Particularly, the predicates at 0x401a4c and 0x4094fc should take the `true` branch and `false` branch respectively, the jump table at 0x40a322 should take the entry at 0x40a566. Currently, PMP supports ELF binary on the x86_64 platform. It can be easily extended to support other architectures due to the cross-platform feature of QEMU. We leave it as our future work. In the rest of the subsection, we discuss a number of practical challenges faced by PMP.

Handling File and Network I/O, Infinite Loop and Recursion. Forced execution may result in exceptional program behaviors, such as invalid file/network access, infinite loop and infinite recursion. To make PMP applicable to real-world executables, these issues need to be handled. PMP follows similar solutions to X-Force regarding these problems. The difference lies in that we implement them on QEMU while X-Force was on PIN. We briefly discuss these solutions for the completeness of discussion.

To handle invalid file access, PMP wraps file open functions (e.g., `open` and `fopen`). If the file to be opened does not exist, a file padded with random values will be used. To handle infinite loop, PMP adopts the profiling-based approach proposed in [31] to dynamically identify loop structures. For each identified loop structure, PMP resets the loop bound to a pre-define constant. This is more sophisticated than X-Force, which uses a fixed global loop bound. To handle infinite recursion, PMP intercepts call and return instructions to maintain a call stack. At each function invocation, PMP checks whether the appearances of the target function in the call stack exceed a pre-defined threshold. If so, PMP skips the function invocation. Note that while maintaining a faithful shadow call stack is very challenging due to the various strange calling conventions, PMP does not require a precise shadow stack.

Allocation of Large PAMA. PAMA is located at the lower part of the address space starting from 0x0. The default load address for non-position-independent executables is usually 0x400000. If the size of PAMA is larger than 4MB, there will be overlap between PAMA and the text/data segment of the subject executable, which is problematic.

To support large-size PAMA, we enable the address mapping mechanism provided by QEMU, which translates a guest address (denoted as `GA`) used by the subject executable to a host address (denoted as `HA`) used by QEMU. In the user-mode emulation, QEMU and the subject executable share the same address space. The address mapping $g2h$ is flattened to essentially an offsetting operation, such that $ha = g2h(ga) = ga + base$, where $ga \in \texttt{GA}$, $ha \in \texttt{HA}$, and $base$ is a pre-defined base address. We set the base address to the size of PAMA to avoid any overlap. Consequently, we need to adjust the filling values accordingly such that they are mapped to the addresses within PAMA (started from 0x0 in the host space). Formally, let $\texttt{FV}^{'}$ be the set of the adjusted filling values. Then we have $\texttt{FV}^{'} = \{x - base \mid x \in \texttt{FV}\}$.

Misaligned Memory Access. The memory pre-planning of PMP assumes that any pointer field of a structure is word-aligned. It is a reasonable assumption for most real-world applications, since making pointer fields word-aligned (by padding if needed) is the default behavior of compilers. For example, mainstream compilers will place a 7-byte padding between the `f3` field and the `f4` field of the structure `G` in Figure 5a by default, such that the offset of `f4` is word-aligned.

Although we didn't find any real-world cases in our evaluation, it is possible to disable word-alignment via a special compilation option. The misalignment of a pointer field (within PAMA) may result in invalid memory access. For example, assume the global variable `g` in Figure 5a points to 0xfff0 set by PMP. If its pointer field `f4` is not word-aligned, its value will be loaded from 0xfff1, which would be 0xe800000000000050. If this value is used as an address, the access falls out of PAMA (even out of the user address space) and causes exception.

We develop the following mechanism in the dispatcher to handle misaligned memory accesses in a demand driven fashion. If a path scheme results in invalid memory access in all the executors (most likely induced by misaligned accesses), the dispatcher checks the QEMU exception log to acquire the instruction i that accesses misaligned address. Then PMP additionally intercepts the code generation of instruction i to mask the most-significant bytes of the accessed memory address to make it fall within PAMA. Note that while our design anticipates misaligned pointer field accesses are rare, which is true according to our experience (see Section IV), it is possible future malware may purposely introduce lots of such misalignments. In this case, PMP would have to instrument all memory operations to sanitize the addresses.

IV. EVALUATION

A. Experiment Setup

We evaluate PMP with the SPEC2000 benchmark set as well as a set of malware samples provided by VirusTotal [12] and Padawan [8]. The experiment on SPEC2000 is conducted on a desktop computer equipped with an 8-core CPU (Intel® Core™ i7-8700 @ 3.20GHz) and 16G main memory. The experiment on the malware samples is conducted on a virtual machine (to sandbox their malicious behaviors) hosted on the same desktop. On both experiments, the configuration of PMP is as follows: 4-MB pre-allocated memory area (i.e., $S = 0\text{x}400000$), diversity $d = 1$, and 2 executors (i.e., $n = 2$).

B. SPEC2000

SPEC2000 is a well-known benchmark set contains 12 real world programs, some of them are large (e.g., *176.gcc*). The list of programs and the characteristics of their executables can be found in Appendix A. We choose SPEC2000 for the purpose of comparison as it was used in X-Force. Table I presents the comparative results on different aspects, including forced execution outcomes, code coverage and memory dependence.

Forced Execution. In this experiment, both PMP and X-Force use the same linear path exploration strategy. Specifically, it first executes the binary once without forcing any branch outcome. Then it traverses the executed predicates in the reverse temporal order (the last predicate first) and finds the predicate that has an uncovered branch. A new path scheme is then generated to force-set the uncovered branch. The procedure repeats until there are no more schemes that can lead to new coverage. Column 2 in Table I reports the total execution time when PMP finishes the exploration. Columns 3 and 4 present the number of executions that pass and fail (i.e., encounters an exception), respectively. The number in parentheses denote the number of executions finished per second. Columns 11-13 show the corresponding results for X-Force. From these results, we have the following observations. (1) PMP can perform 12.6 forced executions per second on average, which is 84 times faster than X-Force (0.15 execution per second). Since PMP uses 2 executors for each path scheme, one may argue that X-Force can be parallelized to use two cores (for fair comparison). We want to point out that first it is unclear how to parallelize the linear search algorithm; and the second executor in PMP is just to provide better probabilistic guarantees. In most cases, such improvement may not have practical impact (see our next experiment). Hence in deployment, additional executors may be turned off. (2) The execution failure rate of PMP is 3.5%, which is reasonably low and comparative with X-Force. Note that the rate is higher than what we identified in the SCMB probability analysis (Section III-D). The reason is that the majority of failures reported by both PMP and X-Force are not caused by memory exceptions, but rather inevitable as the path explorer forces the execution to enter branches that must lead to failures (e.g., forcing the true branch of a stack smash check inserted by the compiler).

Code Coverage. Columns 5∼7 and 14∼16 show the code coverage of PMP and X-Force, respectively. Observe that on average PMP covers 83.8% instructions, 79.1% basic blocks and 91.8% functions, which is comparable to X-Force. For most of the benchmark programs, PMP achieves more than 80% code coverage. Specifically, for *mcf* and *gzip*, PMP achieves 100% code coverage.

The worst cases are *eon* and *gcc*. Further manual inspection shows that this is due to some inherent shortcoming of the linear search strategy. To illustrate, consider the code snippet in Figure 6, which is extracted from *gcc* that validates function arguments before proceeding. When the `check_arg()` function is invoked for the first time at line 2, the `true` branch of predicate at line is taken by default. The linear path exploration will force the next execution to take the `false` branch, since it has not been covered before. At the second-time invocation of `check_arg()` at line 3, the `false` branch of the predicate at line 8 will not be forced to execute again (hence take the `true` branch by default), since it has been covered before. That means, the code after line 3 will not get executed due to the validation failure at line 3.

The essence of the problem is that linear search only focuses on predicates, without considering their context. For example, function `check_arg()` may be invoked from multiple places, and each calling context should be considered differently. That is, a branch being covered in a context should not prevent it from being explored again in a different context. In our future work, we will explore a context-sensitive path exploration method that can provide probabilistic guarantees. Specifically, we will explore a sampling algorithm that can sample a predicate, together with its unique context, in a specific distribution (e.g., uniform distribution).

Memory Dependence. We also conducted an experiment, in which we detect the program dependencies exercised by forced execution. A dependence is exercised when an instruction writes to some address, which is later read by another instruction. This is to evaluate the SDMB property of PMP. Note that it is intractable to acquire the ground truth of program dependencies, even with source code (due to reasons such as aliasing). Therefore, we use two methods to evaluate the quality of detected dependencies. First, we run the SPEC programs on the inputs provided by the SPEC suite (some of them are large and comprehensive) and collect the dependencies observed. These must be true positive program dependencies. As such, forced execution is supposed to expose most of them. Any missing one is an FN. Second, we built a static type checker to check if the source and destination of a (detected) dependence must have the same type. We developed an LLVM pass to propagate symbolic information to individual instructions, registers, and memory locations such that we know the type of each binary operation and its operands. Note that we need the symbolic information just for this experiment. PMP operates on stripped binaries. Ideally, force execution should report as few mistyped dependencies as possible. Each mistyped dependence must be an FP. Columns 8∼10 and

TABLE I: SPEC2000 Results

Benchmark	PMP									X-Force								
	execution status			code coverage			memory dependence			execution status			code coverage			memory dependence		
	time (s)	# run	# fail	# insn	# block	# func	# found	# correct	# mistyped	time (s)	# run	# fail	# insn	# block	# func	# found	# correct	# mistyped
164.gzip	24.6	382 (15.6/s)	11 (3%)	7,650 (100%)	699 (99%)	61 (100%)	3,529	2,824 (80%)	0 (0%)	2,112	369 (0.17/s)	10 (3%)	7,420 (97%)	669 (95%)	61 (100%)	3,662	2,343 (64%)	28 (1%)
175.vpr	76.8	1,006 (13.1/s)	82 (8%)	26,783 (83%)	2,007 (71%)	226 (89%)	13,418	8,983 (67%)	333 (2%)	9,436	1,000 (0.10/s)	79 (8%)	26,677 (83%)	2,004 (70%)	226 (89%)	13,332	7,199 (57%)	2,428 (18%)
176.gcc	3490.2	26,524 (7.6/s)	822 (3%)	186,310 (49%)	16,104 (44%)	1,239 (65%)	573,375	384,161 (67%)	11,467 (2%)	347,014	26,647 (0.08/s)	799 (3%)	183,280 (48%)	16,098 (43%)	1,221 (64%)	573,926	332,303 (58%)	63,131 (11%)
181.mcf	8.6	144 (16.7/s)	2 (1%)	2,977 (100%)	213 (100%)	24 (100%)	1,718	1,248 (73%)	0 (0%)	374	164 (0.43/s)	2 (1%)	2,947 (99%)	213 (100%)	24 (100%)	1,487	1,011 (68%)	130 (9%)
186.crafty	860.3	2,753 (3.2/s)	15 (0.5%)	40,404 (96%)	4,237 (96%)	104 (100%)	22,437	14,300 (64%)	20 (0.08%)	99,764	2,830 (0.03/s)	13 (0.4%)	41,685 (99%)	4,381 (99%)	104 (100%)	22,816	12,092 (53%)	2,749 (12%)
197.parser	98.2	1,590 (16.2/s)	68 (4%)	22,093 (90%)	2,688 (92%)	279 (94%)	9,958	6,664 (67%)	887 (9%)	6,340	1,685 (0.27/s)	69 (4%)	23,331 (95%)	2,799 (96%)	288 (97%)	11,740	5,870 (50%)	3,682 (31%)
252.eon	37.2	707 (19.0/s)	27 (4%)	28,600 (71%)	5,560 (70%)	502 (82%)	9,521	4,457 (47%)	142 (1%)	4,020	659 (0.16/s)	26 (4%)	27,622 (69%)	5,413 (68%)	501 (81%)	9,121	3,557 (39%)	5,669 (62%)
253.perlbmk	1,189	10,318 (8.7/s)	508 (5%)	118,135 (88%)	11,600 (90%)	692 (97%)	66,726	28,394 (43%)	4,001 (6%)	176,096	10,400 (0.06/s)	502 (4%)	119,467 (89%)	11,676 (90%)	696 (97%)	70,611	24,713 (35%)	18,866 (27%)
254.gap	1,054	7,754 (7.3/s)	310 (4%)	49,869 (54%)	4,519 (50%)	401 (88%)	38,243	20651 (54%)	3,059 (8%)	103,458	7,461 (0.07/s)	298 (4%)	49,920 (54%)	4,521 (50%)	401 (88%)	38,784	18228 (47%)	6,593 (17%)
255.vortex	487.0	7,232 (14.9/s)	157 (2%)	100,718 (92%)	15,513 (91%)	577 (92%)	55,205	19,939 (36%)	630 (1%)	58,646	7,223 (0.12/s)	132 (2%)	100,652 (92%)	15,489 (91%)	577 (92%)	54,977	15,393 (28%)	14,072 (26%)
256.bzip2	16.0	249 (15.6/s)	13 (5%)	6,338 (92%)	545 (94%)	60 (95%)	2,755	2,375 (86%)	0 (0%)	842	258 (0.19/s)	11 (4%)	5,179 (76%)	471 (82%)	53 (84%)	2,434	1,849 (76%)	215 (9%)
300.twolf	221.4	2,972 (13.4/s)	97 (3%)	52,351 (91%)	3,682 (86%)	165 (99%)	24,032	10,333 (43%)	528 (2%)	21,308	2,997 (0.14/s)	90 (3%)	52,831 (92%)	3,749 (88%)	165 (99%)	25,664	8,212 (32%)	3,132 (12%)
Average	-	12.6/s	3.5%	83.8%	79.1%	91.8%	-	60.6%	2.6%	-	0.15/s	3.4%	82.7%	81.0%	90.9%	-	50.6%	19.6%

```
01 int some_func(char *arg1, char *arg2) {
02   check_arg(arg1);
03   check_arg(arg2);
04   do_something(); // do nothing
05   ...
06 }
07 void check_arg(char *arg) {
08   if (strlen(arg) == 0) exit(-1);
09   ...
10 }
```

Fig. 6: Explaining problem of linear search using *gcc*.

```
01 long suspend_impl(..){..
02   if (is_valid(arc)) {..
03     memcpy(new_arc, arc, 0x40);..
04     *(arc->tail) = node1;..
05     node2 = *(new_arc->tail);..
06   }
07 }
```

Fig. 7: Explaining FPs and FNs by X-Force using *mcf*.

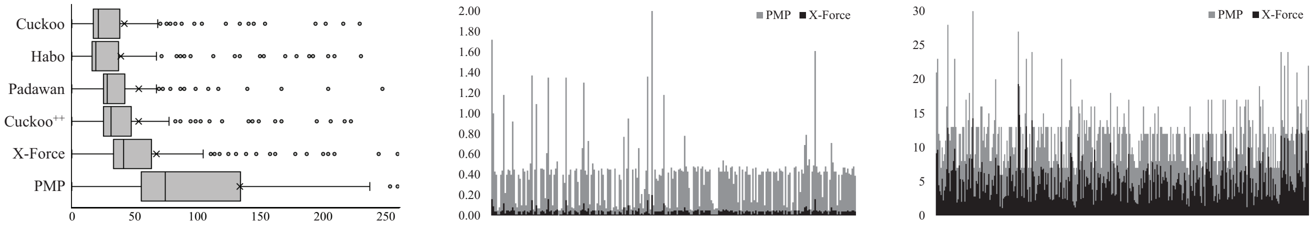
17∼19 show the memory dependence results for PMP and X-Force, respectively.

Observe that X-Force has 6.5 times more mis-typed memory dependences compared to PMP (19.6% versus 2.6%), that is, 6.5X more FPs. In addition, the must-be-true memory dependences reported by X-Force are 10% fewer than those by PMP. That is, X-Force has 10% more FNs. The main reason is that X-Force does not trace into library execution such that pointer relations are incomplete. We will use a case study to explain this in the next paragraph. Mis-typed dependences (FPs) in PMP are mostly caused by violations of SDMB. The results are consistent with our analysis in Section III-D. Note that our probabilistic guarantee for SDMB was computed for a pair of accesses, whereas the reported value is the expected value over a large number of pairs.

Case Study. We use *181.mcf* as a case study to demonstrate the advantages of PMP over X-Force, as well as over a naive memory pre-planning that fills the pre-allocated region and variables with 0. To reduce the interference caused by the path exploration algorithm, we use the execution traces of the runs on the provided test cases as the path schemes. That is, we enforce the branch outcomes in a way that strictly follows the traces. The test cases fall into three categories: *training*, *test*, and *reference*, with difference sizes (reference tests are the largest). We use the memory dependences reported while executing the test cases normally as the ground truth to identify the false positives and false negatives for PMP and X-Force. Since both the forced and unforced executions of a test input follow the same path, the comparison particularly measures the effectiveness of the memory schemes. To be more fair, we only run PMP on a single executor.

The results are shown in Table II. The 2nd and 3rd columns compare the execution speed. Observe that PMP is much faster, consistent with our earlier observation. For the memory dependences, PMP has no FPs or FNs while the naive planning method has some; and X-Force has the largest number of FPs and FNs. The former is because SDMB is violated. The latter is due to the incompleteness of pointer relation tracking (i.e., missing the library part). Note that the numbers of FPs and FNs are smaller compared to the previous experiment as these are results for a small number of runs, without exploring paths.

Consider the code snippet from *mcf* shown in Figure 7. Variable `arc` is a buffer that contains many pointer fields. As it is copied to `new_arc` at line 3, the pointer fields in `arc` and `new_arc` are linearly correlated. However, X-Force misses such correlations as it does not trace into `memcpy()` at line 2. This could lead to missing dependences such as that between lines 4 and 5; and also bogus dependences. For example, the read `*(new_arc->tail)` at line 5 must falsely depend on some write that happened earlier.



(a) number of exposed syscall sequences. (b) executions per second. (c) length of path scheme.

Fig. 8: Overall result of malware analysis.

TABLE II: Experiment with *mcf*.

Item	Execution Time (s)		Memory Dependence										
	PMP	X-Force	ground	PMP			Naive			X-Force		
				found	fp	fn	found	fp	fn	found	fp	fn
test	0.0305	1.987	1847	1847	0	0	1848	5	4	1858	28	17
train	0.0348	2.578	2065	2065	0	0	2069	13	9	2088	45	22
ref	0.0609	4.390	2062	2062	0	0	2068	14	8	2080	37	19

C. Malware Analysis

We use 400 malware samples. Half of them are acquired from VirusTotal under an academic license, and the other half fall into the set of malware used in the Padawan project. Note that the authors of Padawan cannot share their samples due to licensing limitations. Hence, we crawled the Internet for these samples based on a set of hash values provided by the Padawan's authors through personal communication. Many samples could not be found and are hence elided. The 400 samples cover up-to-date malware of different families captured from year 2016 to 2018. We compare the malware analysis result of PMP with that of Cuckoo [2] (a well-known sandbox for automatic malware analysis), Padawan [8] (an academic multi-architecture ELF malware analysis platform), Habo [10] (a commercial malware analysis platform used by VirusTotal for capturing behaviors of ELF malware samples) as well as X-Force [32].

In order to compare our technique with the state-of-the-art anti-evasion measures, we implemented two popular anti-evasion methods [19] (i.e. system time fast-forwarding and anti-virtualization-detection) as extensions to Cuckoo. We name the extended system Cuckoo^{++}. Specifically in the first method, we modify the kernel to make the system clock much faster (e.g., 100 times faster), mainly for the following two reasons. First, a malware analysis VM often has a very short uptime since it restarts for each malware execution. As such, advanced malware may check the system uptime to determine the presence of sandbox VM. Second, advanced malware samples often sleep for a period of time before executing their payload (in order to defeat dynamic analysis). In the other method, we intercept file system operations to conceal the artifacts produced by virtual machine (e.g., /sys/class/dmi/id/product_name and /sys/class/dmi/id/sys_vendor).

The detailed comparison results are shown in Appendix C. Note that the malware behaviors of Padawan are provided by its authors. We set up an execution environment similar to Padawan (Ubuntu 16.04 with Linux kernel version 4.4) for the other tools, including PMP, X-Force, Habo, Cuckoo and Cuckoo^{++}, so that the results can be comparable. We set 5 minutes timeout for each malware sample.

TABLE III: Analysis on malware samples used for case study.

Case	ID	Cuckoo	Habo	Padawan	Cuckoo++	X-Force	PMP
1	031	12	17	12	12	283	301
2	004	27	29	28	27	32	216
3	225	49	49	166	165	183	220
4	309	153	169	292	221	274	705

Result Summary. Figure 8 presents the overall result of malware analysis. Specifically, the number of unique system call sequences exposed by different tools are show in Figure 8a. To avoid considering similar system call sequences that have only small differences on argument values as different sequences, we consider sequences that have more than 90% similarity as identical. As we can see that the executions with anti-evasion measures enabled (i.e., Cuckoo^{++} and Padawan) expose more system call sequences than the native executions (i.e., Cuckoo and Habo), but disclose fewer than the forced execution methods (i.e., X-Force and PMP). On average, PMP reports 220%, 243%, 150%, 151% and 98% more system call sequences over Cuckoo, Habo, Cuckoo^{++}, Padawan and X-Force, respectively. Details can be found in Appendix C.

The comparison of execution speed and length of path schemes between PMP and X-Force are shown in Figure 8b and Figure 8c respectively. Note that Cuckoo and Padawan only runs each sample once (instead of multiple executions on different path schemes as force execution tools do). Hence we do not compare their execution speeds and length of path scheme. On average, PMP is 9.8 times faster than X-Force and yields path schemes with the length 1.5 times longer than X-Force. The longer the path scheme, the deeper the code was explored. The second case studies in this subsection show that with the longer path schemes, PMP can expose some malicious behavior in deep program paths that could not be exposed by X-Force.

Case Studies. Next, we use four case studies from different malware families to illustrate the advantages of PMP.

Case1: `1e19b857a5f5a9680555fa9623a88e99`. It is a ransom malware that uses UPX packer [11] to pack its malicious payload in order to evade static analysis. Figure 9a shows a constructed code snippet to demonstrate part of its malicious logic. It mmaps a writable and executable memory area (line 2), then unpacks itself (line 3) and transfers control

```
01 int main(int argc, char **argv) {
02   void *code_area = map_exec_write_mem();
03   upx_unpack(code_area);
04   transfer_control(code_area, argc, argv);
05 }
06
07 void code_area(int argc, char **argv) {
08   if (!is_cmdline_valid(argc, argv)) exit();
09   char *action = argv[1], *key = argv[2];
10   delete_self();
11   if (strcmp(action, encrypt) == 0) {
12     for (FILE *file: traverse_directory()) {
13       FILE *encrypted_file = encrypt(file, key);
14       replace_file(encrypted_file, file);
15     }
16   }
17 }
```

(a) simplified code.

```
a. mmap(0x400000,,PROT_EXEC|PROT_READ|PROT_WRITE,)
b. unlink("/root/Malware/1e19b857a5f5a9680555fa9623a88e99")
c. open("/etc",O_RDONLY|O_DIRECTORY|O_CLOEXEC)
d. getdents64(0,)
e. open("/etc/passwd",O_RDONLY)
f. open("/etc/passwd.encrypted",O_WRONLY|O_CREAT,0666)
g. unlink("/etc/passwd")
```

(b) captured system call sequence.

Fig. 9: Case 1: the ransom malware sample.

(line 4) to the unpacked payload (lines 7-17). The malicious payload checks the validity of command line parameters (line 8) and deletes itself from the file system (line 10). If the command line parameter specifies the `encrypt` action, the malware traverses the file system to replace each file with its encrypted copy (lines 13-14).

The comparison of different tools on this malware is shown in the second row of Table III. Triggering payload requires the correct command line parameters. Hence directly running the malware using Cuckoo, Habo, Cuckoo^{++} and Padawan fail to expose the malicious behavior. Both X-Force and PMP expose the payload. Figure 9b shows the captured system call sequence. Observe the `unlink` syscall `b` that removes the malware itself and the encryption and removal of "`/etc/passwd`" by syscalls `e-g`.

Case2: `03cfe768a8b4ffbe0bb0fdef986389dc`. It is a bot malware that receives command from a remote server. Figure 10a shows the simplified code of its processing logic. It checks whether a file exists that indicates the right execution environment (line 2) and whether the remote server is connectable (line 4). If both conditions are satisfied, the malware communicates with the remote server. The remote server will validate the identity of the malware by its own communication protocol (lines 4-7). If the validation is successful, a command received from the remote server will be executed on the victim machine (lines 8-9).

The comparison of different tools on this malware is shown in the third row of Table III. The malicious payload of this malware sample is hidden in a deeper path, which requires a much longer path scheme. Figure 10b shows the path scheme enforced by PMP to expose the malicious behaviors. The length is 28, which is larger than the longest path scheme that is enforced by X-Force within the 5 minutes limit. These forced branches are to get through the ID validation protocol.

```
01 int main(int argc, char **argv) {
02   if (!files_exist("/tmp/ReV1112")) exit(0);
03   if (!connectable("ka3ek.com")) exit(0);
04   Info *info = get_system_info();
05   Greet *greet = get_validation(info);
06   Reply *reply = compute_reply(greet);
07   Cmd *cmd = get_command(reply);
08   if (!cmd) exit(0);
09   execute_cmd(cmd);
10 }
```

(a) simplified code.

```
40492b:T | 404aec:T | 404e07:T | 401f3f:F | 401ee3:T |
404fdc:F | 404fea:T | 405118:F | 40513a:F | 405144:F |
40517b:F | 40517f:F | 40523e:F | 405254:T | 40523e:F |
405254:T | 40523e:F | 405254:T | 40523e:F | 405254:T |
40523e:F | 405254:F | 4044be:T | 4044e9:F | 40454b:F |
404565:T | 404596:T | 404794:F
```

(b) path scheme.

Fig. 10: Case 2: the bot malware sample.

Case3: `14b788d4c5556fe98bd767cd10ac53ca`. It is an enhanced variant of Mirai, which is equipped with a time-based cloaking technique. Figure 11 shows a simplified version of its code snippet. At line 4, it checks whether the system uptime is short, which indicates a potential analysis environment. If the system uptime is long enough, it checks whether there exists any initialization script in the "/etc/init.d" directory (line 8) [2]. If both conditions are satisfied, the malware sample adds itself to an initialization script for launching at system reboot.

Cuckoo and Habo cannot expose the aforementioned behaviors. Cuckoo^{++} and Padawan can expose the traversal of the "/etc/init.d" directory (line 6), by passing though the uptime check via fast-forwarding system time and using a long-running VM snapshot, respectively. However, they cannot expose the modification of initialization script (line 9), due to the failure of the initialization script check, as the default OS environment does not have any initialization script. PMP and X-Force can expose both behaviors by forcing the branch results.

Case4: `8ab6624385a7504e1387683b04c5f97a`. This is a sniffer equipped with a vm-detection-based cloaking technique. Figure 12 shows a simplified version of its code snippet. If a VM environment is detected, the malware sample deletes itself and exits (lines 2-3). Otherwise, it enters a sniffing loop, which randomly selects an intranet IP address and a known vulnerability and checks whether the host with the IP contains the vulnerability (lines 5-7). If so, the information about the vulnerable host is sent to the server and the payload is sent to the vulnerable host (lines 8-9).

Cuckoo and Habo cannot expose the aforementioned behaviors. Cuckoo^{++} and Padawan can expose the network communication to the selected IP address, since they are enhanced to conceal VM-generated artifacts. However, they cannot expose sending the vulnerable host information and payload, since the analysis environment is often offline and there may not exist a vulnerable host on the intranet. PMP can expose both behaviors. X-Force can expose both in theory

---

[2] An initialization script has a file name that starts with 'S', followed by a number indicating the priority.

```
int main(int argc, char **argv) {
  struct sysinfo info;
  sysinfo(&info);
  if (info.uptime < 128) exit(0);
  DIR *dir = opendir("/etc/init.d");
  while (struct dirent *ent = readdir(dir)) {
    char name = ent->d_name;
    if (name[0] == ’S’ && is_num(name[1]))
      add_to_init_script("/etc/init.d/S99");
  }
}
```

Fig. 11: Case 3: the enhanced variant of Mirai.

```
char *data = read_file("/sys/class/dmi/id/product_name");
if (contains(data, "VirtualBox", "VMware"))
  remove_self_and_exit();
while (1) {
  char *ip = select_intranet_ip(ip_list);
  char *vuln = select_known_vuln(vuln_list);
  if (connect_and_check(ip, vuln)) {
    send_info_to_server(ip, vuln);
    send_payload(ip, vuln);
  }
}
```

Fig. 12: Case 4: the sniffer malware sample.

but fails within the timeout limit due to its substantially larger runtime cost.

D. Time Distribution

We measure the runtime overhead of different components. The distribution is shown in Appendix B. As we can see that most of the time (84%) is spent on code execution, while only 13% and 3% of time are spent on memory pre-planning and path exploration, respectively. In memory pre-planning, 2%, 5%, 69% and 24% of time are spent on PAMA preparation, initialization of global variables, local variables and heap variables. Observe that PAMA preparation takes very little time as most work is done offline.

V. RELATED WORK

Forced Execution. Most related to our work is X-Force [32]. The technical differences between the two were discussed in the introduction section. As shown by our results, PMP is 84 times faster than X-Force, has 6.5X, and 10% fewer FPs and FNs of dependencies, respectively, and exposes 98% more payload in malware analysis. Following X-Force, other forced-execution tools are developed for different platforms, including Android runtime [33] and JavaScript engine [25], [21]. Compared to these techniques, PMP targets x86 binaries and addresses the low level invalid memory operations. Additionally, PMP is based on novel probabilistic memory pre-planning instead of demand driven recovery.

Memory Randomization. Memory randomization has been leveraged for different purposes, such as reducing vulnerability to heap-based security attacks through randomizing the base address of heap regions [14] and randomly padding allocation requests [15]. DieHard [13] tolerates memory errors in applications written in unsafe languages through replication and randomization. It features a randomized memory manager that randomizes objects in a “conceptual heap” whose size is a multiple of the maximum real size allowed. PMP shares a similar probabilistic flavor to DieHard. The difference lies in that PMP pre-plans the memory by pre-allocation and filling the pre-allocated space and variables with crafted values. In addition, PMP aims to survive memory exceptions caused by forced-execution whereas DieHard is for regular execution.

Malware Analysis. The proliferation of Malware in the past decades provide strong motivation for research on detecting, analyzing and preventing malware, on various platforms such as Windows [16], [23], Linux [19], [20], as well as Web browsers [24], [22]. Traditional malware analysis fall into two categories: signature-based scanning and behavioral-based analysis. The former [12], [28] detects malware by matching extracted features with known signatures. Although commonly used by anti-malware industry, signature-based approaches are susceptible to evasion through obfuscation. To address this, behavioral-based approaches [34], [26], [17] execute a subject program and monitor its behavior to observe any malicious behavior. However, traditional behavioral-based approaches are limited to observing code that is actually executed.

Anti-targeted Evasion. Modern sophisticated malware samples are equipped with various cloaking techniques (e.g., stalling loop [27] and VM detection [6]) to evade detection. To fight against evasion, unpacking techniques [18], [29] are applied to enhance signature-based scanning, and dynamic anti-evasion methods [26], [30] are developed to hide dynamic features of analysis environment such as execution time and file system artifacts. These techniques are very effective for known targeted evasion methods. Compared to these techniques, PMP is more general. More importantly, PMP and forced execution type of techniques allow exposing payload guarded by complex conditions that are irrelevant to cloaking.

VI. CONCLUSION

We develop a lightweight and practical force-execution technique that features a novel memory pre-planning method. Before execution, the pre-planning stage pre-allocates a memory region and initializes it (and also variables in the subject binary) with carefully crafted values in a random fashion. As a result, our technique provides strong probabilistic guarantees to avoid crashes and state corruptions. We apply the prototype PMP to SPEC2000 and 400 recent malware samples. Our results show that PMP is substantially more efficient and effective than the state-of-the-art.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers and Dr. William Robertson (the PC contact) for their constructive comments. Also, the authors would like to express their thanks to VirusTotal and the authors of Padawan for their kindness in sharing malware samples and the analysis results. The Purdue authors were supported in part by DARPA FA8650-15-C-7562, NSF 1748764, 1901242 and 1910300, ONR N000141410468 and N000141712947, and Sandia National Lab under award 1701331. The UVA author was supported in part by NSF 1850392.

REFERENCES

[1] Clickless powerpoint malware installs when users hover over a link. https://blog.barkly.com/powerpoint-malware-installs-when-users-hover-over-a-link.

[2] Cuckoo. https://cuckoosandbox.org/.

[3] Cybersecurity statistics. https://blog.alertlogic.com/10-must-know-2018-cybersecurity-statistics/.

[4] Evil clone attack. https://gbhackers.com/evil-clone-attack-legitimate-pdf-software.

[5] Fileless malware. https://www.cybereason.com/blog/fileless-malware.

[6] Linux anti-vm. https://www.ekkosec.com/blog/2018/3/15/linux-anti-vm-how-does-linux-malware-detect-running-in-a-virtual-machine-.

[7] Mirai malware. https://en.wikipedia.org/wiki/Mirai_(malware).

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

[9] Qemu user emulation. https://wiki.debian.org/QemuUserEmulation.

[10] Tencent habo. https://blog.virustotal.com/2017/11/malware-analysis-sandbox-aggregation.html.

[11] Upx. https://upx.github.io/.

[12] Virustotal. https://www.virustotal.com/gui/home/upload.

[13] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06. ACM, 2006.

[14] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a board range of memory error exploits. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03. USENIX Association, 2003.

[15] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05. USENIX Association, 2005.

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

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

[18] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. 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*. ACM, 2018.

[19] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding linux malware. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*, 2018.

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

[21] Xunchao Hu, Yao Cheng, Yue Duan, Andrew Henderson, and Heng Yin. Jsforce: A forced execution engine formalicious javascript detection. In Xiaodong Lin, Ali Ghorbani, Kui Ren, Sencun Zhu, and Aiqing Zhang, editors, *Security and Privacy in Communication Networks*. Springer International Publishing, 2018.

[22] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. Hulk: Eliciting malicious behavior in browser extensions. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014.

[23] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. Cutting the gordian knot: A look under the hood of ransomware attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015.

[24] Amin Kharraz, William Robertson, and Engin Kirda. Surveylance: automatically detecting online survey scams. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018.

[25] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. J-force: Forced execution on javascript. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17. International World Wide Web Conferences Steering Committee, 2017.

[26] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and Xiaofeng Wang. Effective and efficient malware detection at the end host. In *USENIX 2009, 18th Usenix Security Symposium*, 2009.

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

[28] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2005.

[29] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *23rd Annual Computer Security Applications Conference (ACSAC 2007)*, 2007.

[30] Kirti Mathur and Saroj Hiranwal. A survey on techniques in detection and analyzing malware executables. *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(4), 2013.

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

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

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

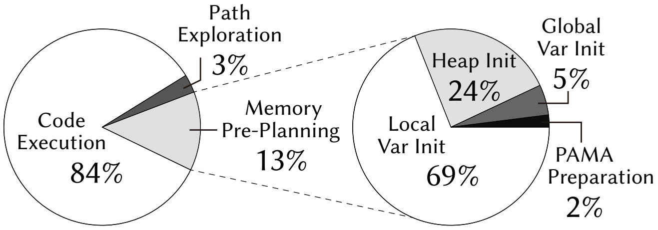
[34] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07. ACM, 2007.

APPENDIX

A. *Spec2000 Benchmark*

Benchmark	source lines	binary size	# insn	# block	# func
164.gzip	8,643	143,760	7,650	707	61
175.vpr	17,760	435,888	32,218	2,845	255
176.gcc	230,532	4,709,664	378,261	36,931	1,899
181.mcf	2,451	62,968	2,977	213	24
186.crafty	21,195	517,952	42,084	4,433	104
197.parser	11,421	367,384	24,584	2,911	297
252.eon	41,188	3,423,984	40,119	7,963	615
253.perlbmk	87,070	1,904,632	133,755	12,933	717
254.gap	71,461	1,702,848	91,608	9,020	458
255.vortex	67,257	1,793,360	109,739	16,970	624
256.bzip2	4,675	108,872	6,859	577	63
300.twolf	20,500	753,544	57,460	4,280	167

B. *Time Distribution*



C. *Details of Malware Analysis Result*

	Cuckoo	Habo	Padawan	Cuckoo++	X-Force	PMP
Avg.	41.65	38.88	53.15	53.28	67.40	133.36

ID	MD5	Cuckoo	Habo	Padawan	Cuckoo++	X-Force	PMP
001	00056adfd6982498c184f429d7af61d4	29	22	28	31	42	92
002	005449f26bb0033c8ba5cfbb5c2c6f6b	15	15	25	27	34	74
003	0191642afcabb6cb2e9449822ea10d37	70	65	69	98	126	128
004	03cfe768a8b4ffbe0bb0fdef986389dc	27	29	28	27	32	216
005	045136430edac124ea134bf2a32a4a60	15	15	26	15	16	33
006	057857302490521bd52d25a141bbbdfb	14	14	27	27	34	591
007	0686a7459152174f821c8c635cfbda8a	47	53	47	49	67	90
008	08afb6111b6b3d574036cf10fe787063	14	14	25	27	33	45
009	09e4b26df6b499a81453766c17226106	22	32	27	37	46	80
010	0b855d8d6a3c3ac8d5fd6931570e02ae	48	53	48	55	69	92
011	0bc2cbb5be3e651355a50c07885464bf	15	15	26	29	17	33
012	0c0d2ed33316dc5a92a2785007dbcb50	7	10	17	19	25	27
013	0c1aa91e8cae4352eb16d93f17c0da2b	15	15	25	23	34	74
014	0cfe8985c56da5a821ff9bf35aa3dbd4	22	35	30	35	44	58
015	0d186ccf5829dd5bffdc2aff944fe2f6	23	21	34	37	45	61
016	10c47191922eefcfae39bf5be540bd44	15	16	23	27	35	35
017	10f5beac257a92665866cdc99550b7bb	20	17	19	25	31	347
018	113c079464639b4a12826b42c1d96ac7	24	22	35	35	46	71
019	11c489ddea858030b23f7ac184994439	48	54	47	55	69	87
020	1226e436e5e830c9fbe58043fa4f9f3b	43	40	42	59	76	83
021	1321bd12e164aa7c8b7e39afe7bc8a62	20	18	31	27	42	56
022	132397a7e793fb4052f8d44634a15582	36	40	36	50	63	73
023	137c1520b37dfc3ce5072be7995c96fc	14	14	24	26	33	45
024	13f2bb2af16f513b4a35a26c6f8f5cbc	40	41	42	40	58	64
025	17579313f14995e2bfa75a703562debf	17	18	27	26	40	56
026	179c7648bb607147973c2fccbcc0e530	21	25	31	34	43	43
027	199c8ffc248a35d99e1f26ff79bd9398	14	20	14	14	18	32
028	1a7e8ddc317806db053c472e1299fe33	15	15	25	26	34	74
029	1b5054939ee601d89fdaa44c109943cf	29	22	28	29	42	91
030	1b74e8a749948d2fbf2f90486ce63fcf	17	18	27	30	40	55
031	1e19b857a5f5a9680555fa9623a88e99	12	17	12	12	283	301
032	2077166b21e9717df706ca897e5bfc94	14	14	24	14	15	44
033	210e4243c8edc87499ce7caa4076d433	22	45	41	40	60	69
034	22dc1db1a876721727cca37c21d31655	5	8	5	7	17	135
035	23c42760532270113de57b97346edff0	20	18	30	30	42	56
036	24bf1279bc8ffe0c8380675cb8c1b94a	17	17	27	30	39	40
037	25c364af9d8025dcaa8f6ac10c8283af	17	18	28	32	40	55
038	28255eb4c29ef0420572126d8bc0e481	20	18	30	30	42	54
039	28c866843a9462113eb26aef1024db08	17	17	28	29	36	55
040	28fed854eeadd32abfd946e0692c9ae4	21	19	32	33	42	42
041	2ad28d994083eb88d56eded361d7e381	22	45	40	41	53	80
042	2d66f629e00042de8662b384b3c7c3bb	20	30	25	24	43	46
043	2e6453a7eac407dbe47b70b72082490c	20	18	30	33	42	56
044	31c55141129151ee4728a40613b93eca	21	17	21	21	31	55
045	3544c1e682d97dc5e5dbef6898f17fcf	17	18	28	32	40	55
046	36263d91d726dcdb93b97ea05ae8656a	36	40	36	36	63	69
047	36a332f5a8dc058fdf437fa67ecc06cf	39	36	38	40	58	74
048	39d46a0cd60393e5571b720c915db30d	48	54	47	54	69	93
049	3ad6f8a257cfa2d11292cb6420ed884a	18	19	28	26	41	57
050	3b0d923cf1792151e6540ca38b3d6d19	20	17	19	20	32	74
051	3decf1b4e5e821c159e051a04fbf0452	7	7	20	19	27	27
052	3e21a608b64341e97a73861fa0b24ec2	20	18	32	49	61	75
053	3f193286767c269b786e117c43807f7b	17	18	27	32	40	55
054	3fb857173602653861b4d0547a49b395	14	14	27	27	34	487
055	40845a4a9024e1a44bf2453c11dc4003	18	18	29	28	41	55
056	4087376ef72170f248eb2f0665a26796	22	19	22	25	33	40
057	424f94d07b45eab1bd32494cdeb4d67b	22	45	40	35	22	66
058	46eaf3f07c2a59e0bb284a7aacb41dc4	40	37	39	44	62	136
059	483b322b42835227d98f523f9df5c6fc	30	7	26	30	43	51
060	49c178976c50cf77db3f6234efce5eeb	17	18	27	25	40	40
061	4b1e9e8ccf91998393509290d436ede3	60	61	59	60	77	224
062	4e593af1ab25873681c62ca4f49e31e3	21	19	31	35	43	43
063	4f5d0ed102de7c171d1df4989c4cdcd0	15	16	25	29	36	55
064	4fa4269b7ce44bfce5ef574e6a37c38f	25	16	21	25	37	79
065	502a90ed7a851b01b340aded822c4de0	28	22	27	33	41	119
066	524287dda3d6d8e59ebe249476ed8181	27	23	26	32	42	74
067	53ad943fe07be315d908c6b8fe305a08	24	22	35	40	46	69
068	54b0f140da40e5713377f4d4a8f143ad	24	17	25	26	34	159
069	559169cd8167dcbaaf065d6a122a289d	20	43	38	33	40	57
070	55e0a8737b091da7bda7060b75b2e119	60	61	62	60	101	227
071	56cb1c4e788e63325bbb531da187e609	31	27	59	64	70	96
072	57b1ff91b59aada9a1c566940db4d46a	27	29	28	27	49	90
073	57b4d2108051dbe43d7b35777ba76d40	15	15	26	27	34	78
074	582f47ec975b0ba8cafe5a39cccbd552	24	22	34	36	46	71
075	58af33baf68feb637b59a20ba4ea0c03	26	53	48	51	65	96
076	5a6fd63f4ffc6037dc192b6c3f456e87	55	32	58	58	76	123
077	5b36aebed504b73123e10de21529b638	21	19	31	34	43	43
078	5c1dd20f74dac82306864a411f96171c	140	149	87	140	157	183
079	5c47f09a37376d9b6a4e97518c435dc9	17	18	27	27	39	39
080	5cf6110f21b80123f577e85bf81af82f	22	45	40	47	60	93
081	5d6aa67ce342703f6735925d359c3049	43	40	42	43	77	83
082	5e890cb3f6cba8168d078fdede090996	29	25	28	29	44	76
083	5f13326e2c90b70593b645540f25213f	17	18	28	32	40	55
084	5fb565eee5336c0b30451a0a023036b8	11	5	20	20	29	30
085	5fd2ed4f42f0cce701482fbdb78a00b1	36	7	36	37	39	76
086	5feaa85c62d1117a7931df0bf8b62dd3	21	24	31	33	43	82
087	6025e14c04a7c35e8a049885f035b97b	15	15	23	25	32	32
088	6139657db08c3e9d5d2399259e8eaaa0	28	24	27	33	43	74
089	62c2d296060d14061f5c54f31662dac9	31	27	57	61	88	105
090	6355f0ea6c19090e0baedc57016beb6c	19	20	19	19	31	31
091	664378d10f610552d17e97cc06ade139	20	43	48	39	48	57
092	6b0bd9599779c3a4899a6ee9fd2eee03	28	24	28	35	45	68
093	6dc1f557eac7093ee9e5807385dbcb05	15	15	26	22	34	74
094	705df7bc13a3fc1bbfc79735455fda68	24	21	25	28	34	45
095	70ad6b0a94a0ef3ff974833dd7296b8d	29	25	28	29	45	173
096	717dfa046833dac608b6f1a274a47938	8	7	23	23	29	444
097	72afccb455faa4bc1e5f16ee67c6f915	362	362	291	362	423	505
098	74124dae8fdbb903bece57d5be31246b	36	40	36	38	40	84
099	74f0ec75b6bced0be2ede45455fc90a5	41	7	40	41	44	58
100	75e04ad828359d2d25718430bc5f3dd3	14	14	24	26	33	54

ID	MD5	Cuckoo	Habo	Padawan	Cuckoo++	X-Force	PMP
101	770756fdaed23e4ef3c0a17f26bc22b6	17	17	28	28	35	56
102	7dad01f26f01992d24d0f8e6d08d042e	17	18	28	24	40	55
103	81b6ee216e10e17104706536c21a479a	39	36	39	48	59	157
104	81ea379c237724249c137fc83ef21e9a	6	9	6	6	19	35
105	850177156d5a010254bba5746664a3c7	15	15	26	27	34	34
106	862cfa928c8edfd50ed22e08bbb14c61	17	18	27	25	40	56
107	898dde6afb3142e607528359b0935e9e	48	54	47	48	69	88
108	8bd0c5f36987218a95dc56677c40f880	17	17	27	29	36	57
109	8bfed4ef1067ca119d4d71a66a84e06e	8	7	25	23	29	29
110	8c5c1e62d737ffd0dc36b2c1252ddd75	37	40	38	50	63	266
111	8dba0738910ef34590cea87a3c1ac538	27	32	48	51	74	74
112	8df9ec7cd1de78957ea800fd63d66051	39	36	39	39	58	132
113	8f194847387186899cc8d9f9ca903e07	103	112	139	143	147	190
114	901cbff40784ee40518fda6471e70baa	15	14	26	26	33	70
115	912bca5947944fdcd09e9620d7aa8c4a	15	15	25	26	34	83
116	9353a060cc5fc8f26ce8a0105dfac48f	15	14	26	24	32	34
117	9361a4d5b4bf3041759bd4f727920df2	14	14	28	33	34	587
118	93c2f1ca9949435cffe81572d3d21d5e	15	15	25	27	34	34
119	942ea0c4cb729d4878eb5b8998981228	48	54	48	54	69	92
120	96804156396bce25d49c4ea4f058d569	47	53	48	47	59	83
121	96de2982978ea899ba4a97ff73e7f466	15	15	26	27	34	76
122	97ba48a2562e856d8eef15e1c9f6585e	17	18	28	25	42	57
123	994136a3c18399900f73d085bf42a330	97	94	40	109	138	142
124	9d2b507212c19a9dcf95168745e793ea	39	36	38	39	59	145
125	a25470a5b305fc5e7c80b68810e132b2	43	40	42	46	76	83
126	a27896388f0f0dad493e7d786e48eaab	14	14	23	21	15	95
127	a3ab4dfb3e3b160fed14d923db29daec	20	30	25	23	42	53
128	a4404be67a41f144ea86a7838f357c26	47	43	46	55	69	82
129	a4944230d62083019d13af861b476f33	14	15	24	25	35	54
130	a4eecf76f4c90fb8065800d4cad391df	29	40	59	56	78	303
131	a58fb83be409874271fa04709012b5ad	19	17	29	32	41	55
132	a62f2bca5c0a5d239c6a3732a2f424ab	228	229	359	355	408	621
133	a6617c5cb59135e05799498d264564c7	75	71	72	75	95	129
134	a664df72a34b863fc0a6e04c96866d4c	17	18	28	33	39	172
135	a71079102c6f7053a9402f72cec79825	22	18	21	22	22	33
136	a8cd638e13b1848f347fc724e9386ea8	15	15	23	25	33	79
137	a8f78241bd7b7cad50e054bcb4dfa01b	27	32	48	47	74	74
138	a96fc6e018d771932b70aaf9eb8b7484	347	353	359	415	523	713
139	ab2b936e95da491789caa802ec4948cf	22	19	21	26	34	65
140	ab40bea438fbf809b5786d52b38ea318	39	36	39	39	59	144
141	abbf052d0c9d84c5a30bf7348e225b31	18	19	28	31	40	41
142	ac2c9ce2b3edf07045024d60f9b4e53e	27	32	48	60	75	75
143	ad76e4b7470df9368380b2b5375410b4	40	37	39	49	62	148
144	aec2df8a6cb35aa5b01b0d9f1f879aa1	20	30	25	23	42	49
145	b4088daeb311c24d8f9a20b5ec223bc9	21	17	20	24	31	55
146	b754622e816fb2281402b86f75fa9ccf	26	22	25	26	42	337
147	b8f6cdb7360dd2411fcbed86cf77b775	15	14	25	22	34	34
148	b91fed817500f9c377ca9c799e987c74	27	32	48	60	74	74
149	be0db913011e51e3424be7841b13fd05	15	15	26	26	34	77
150	bf8287805afdfc72ca6b7c6e76d5b04a	347	355	352	351	521	716

ID	MD5	Cuckoo	Habo	Padawan	Cuckoo++	X-Force	PMP
151	c2764861cacf73cda2227bfeb67f707d	8	8	7	10	13	129
152	c2a5b75c7273b3b4d4bf0a234eea35f2	28	24	27	29	43	64
153	c32a5d9b0c78b335af5197d3831966a9	41	42	41	41	50	61
154	c36625389cb4739518472de4298536fb	54	32	58	60	75	120
155	c38d08b904d5e1c7c798e840f1d8f1ee	82	84	87	86	110	137
156	c533142180337d02f5e2a6ee2bf9e099	14	14	27	27	34	587
157	c63cef04d931d8171d0c40b7521855e9	15	15	26	31	34	79
158	c64919c97236dcef4e97140c1153b274	14	7	35	31	46	327
159	c80b8f2a2d6a9e1500bfa52f864ea46d	17	18	28	31	40	55
160	c83b5e8b47824392082c84240bf2f8b4	17	18	27	31	40	55
161	c8c1f2da51fbd0aea60e11a81236c9dc	29	22	28	29	41	91
162	c97acd1fad05a0b0a7825f5647d4244a	17	18	28	32	40	55
163	cb0477445fef9c5f1a5b6689bbfb941e	70	66	70	99	125	127
164	cb3d93f65c64e48ef81274a49a748ce7	29	25	28	36	45	116
165	cc29a224e327412e0db7f3ce5c4f4e00	6	8	6	14	18	35
166	cd60f742fc71f98b34a264c5f3e55a42	14	14	29	25	34	34
167	cfcd5153e739406baa7b354dd5b28e04	17	18	28	31	40	55
168	d04c492a5b78516a7a36cc2e1e8bf521	70	66	69	99	125	127
169	d0874ba34cfbdf714fcf2c0a117cc8e2	38	35	37	38	58	135
170	d0b9d58f3a454ad6df2e4d055858c1e5	6	9	6	14	18	33
171	d1a19e834ef3a4f7ecfcd8af04c6ebe4	14	14	28	23	34	587
172	d21fb7ed52ba13294240354c1f528d2f	3	5	3	10	13	269
173	d2cd482ba82e592c1dc5ded7db79ec70	15	15	25	24	34	74
174	d3a894f6052ecee1ca87b69e619ca0cb	31	27	30	31	49	122
175	d493af745de315c6989355a49d21b2a3	20	18	31	31	42	56
176	d721e7efb5d63eaf85540748942f301d	42	42	43	42	48	52
177	d7d73062d2defe111b6ba3bdcf5e4e18	17	17	28	33	36	55
178	d979d2dce979788c0ce9ccc72b445617	27	32	48	60	74	74
179	dae9fd1c16b6fee713f53182cb2d4e10	17	6	28	31	40	40
180	db16765a02efbe75ae569c5901744c19	346	358	460	465	522	712
181	dc4db38f6d3c1e751dcf06bea072ba9c	15	15	26	29	34	75
182	dd77f74445d61c8d80335b15d432c27b	13	7	21	26	33	110
183	de2e41048e3a54ac1e6bbae91ae999ab	20	43	39	42	30	59
184	de5798b69df92163cdd25f362565c521	27	32	49	44	74	74
185	dff09a1a31fadad518a6760c3cfbdc17	28	46	42	51	64	170
186	e37ff9a3fc89bf29ea96333f3aa7f296	40	37	40	40	60	135
187	e3d80f2cd1de02c74f198189aba33052	29	22	28	29	42	91
188	e6ffa02a63c951e4e8a131e43d9fea6a	15	15	25	20	34	76
189	ec3de1355a2056a7eb5e799b5e989d0b	24	47	44	49	63	92
190	ec673fedd52823da1ebae7019e042382	21	19	31	28	42	82
191	ed62ce1a406b2a0b9d6d79ca4e3572b6	18	18	28	33	37	59
192	ee11c23377f5363193b26dba566b9f5c	31	27	57	43	55	87
193	f27751af292f252f1cc55f90f15bd30b	14	22	284	162	203	423
194	f2b00b27e6e8d10d3c27525ecd9af120	47	53	48	53	69	92
195	f3e8a50f0c1c3a510f882d0fdb121960	14	14	24	26	33	33
196	f8cfc2b7f01c3a26f0a9db32b8c5f51c	17	16	27	27	36	36
197	fa68eb454b37401bb0476428a3ae84a5	20	17	19	24	31	65
198	fa7a3c257428b4c7fda9f6ac67311eda	24	14	20	27	34	159
199	fd75a87293ca3215f3c033f64feefd0f	18	17	29	30	37	58
200	ff02a16427e3200526220350fa8c9b4f	30	26	30	35	44	55

ID	MD5	Cuckoo	Habo	Padawan	Cuckoo++	X-Force	PMP
201	011bb615de58263b483c8fb04d04525c	20	16	19	23	30	519
202	027aaab9a6c3a3d94d78858821555a8b	31	26	30	31	36	107
203	02fc23152110db73763d50fa2c9bf8f9	15	14	42	43	34	70
204	03561dd35406b403d85402979b9d05a2	43	39	42	43	76	82
205	03b2597873ba0f0e28e3dc78343dd968	17	16	28	28	35	52
206	03ed77d8a342473bee100850e42cd11c	7	15	28	20	26	139
207	049d713e7833ac6fa0cdf1b632dce1dd	15	14	26	27	33	70
208	05266ec1f4c9981e7027681563fc8867	59	5	58	59	62	112
209	0632ef98ee12a4754e7c914285625ab0	216	178	299	216	269	344
210	067329430589b374c35e1b696ada34f9	21	23	31	21	27	78
211	06a35dd46bae273bb42850563c9f51fe	38	34	37	45	56	138
212	07ce3c632e2399c1b3218a77599ea771	70	39	99	102	90	260
213	07f5bbc7f414bcb25bbb8014240e8c0f	28	24	27	28	44	56
214	08dbfacee7a4a77f25f159bc8666a974	20	16	19	23	30	563
215	0a44d7078bc1c5f1217ff503f2f3ebc8	8	7	20	17	28	65
216	0b26005c71cea142c87f8e976cf704e0	72	67	89	72	90	222
217	0b9835fd94b8a967497835cb13e212b1	26	17	26	26	33	34
218	0d4de50a28c4294576aa834f13d4f959	15	20	25	15	16	70
219	108079ccf885562a92cb363addb4182c	7	10	17	7	7	138
220	11f6f1bb81a837fab5b578352150a7be	18	18	28	18	23	56
221	125dca58b81561fafe56797252d0a39e	68	63	70	73	71	72
222	135fb83a2a1fad994ac298daa9a427bd	28	23	27	28	42	55
223	13e0645ba42c32bb049419b83f2dc804	17	18	28	30	39	57
224	1408f779af2a5ed4e736af107da29ec8	20	17	19	24	30	46
225	14b788d4c5556fe98bd767cd10ac53ca	49	49	166	165	183	220
226	15b09361380380d3bdcfec7d316b6951	306	306	337	340	350	457
227	196360a06bbef80d5a9aae11f5894a34	20	16	19	23	30	63
228	1a713da3360a34516ad82b1523abf6d1	17	17	28	26	40	64
229	1d21a6d88e50e371e8bde993d7333d89	48	46	47	49	84	86
230	1d5416ae2474aedfd68f79e4aacd1b14	20	17	31	27	41	70
231	1dbfb9de8ddd948039693054fe83459c	78	89	91	83	104	227
232	1ed97c5de81a7a9037727c639faf9bfe	23	20	22	27	34	54
233	1f79632bb62b3497492ec6fa366d98fc	349	407	422	419	495	655
234	21c75019e965cfa6ca34a670c238c379	13	7	22	27	34	147
235	2361605b95afa6514dd856b21854dd26	48	45	47	52	84	86
236	2370ef9dbf483c20f48b4d1a2a6ab3b2	346	208	354	346	360	582
237	256ad86b8cea17b514230497d62b8907	15	15	26	26	23	71
238	25a5284bcd99e246566e0a927fda27fa	17	18	28	31	39	57
239	292d124aa58579e18239951f63c38da7	144	129	205	166	208	300
240	295370e5a3afdb8f6babdfff74837f0b	49	45	53	66	84	85
241	2995574af03023ed9199bdc54de34df0	13	12	24	26	14	38
242	29f518d6fe7de8df6791d110668b912d	29	44	66	69	43	199
243	2b79e388966bb783ba81e56b490f3b93	52	47	58	65	81	141
244	2e940ae965d9ff64a0b225718e765290	21	18	32	33	43	54
245	32370b31ab6b2e23e9ab4add4f2819aa	8	8	20	21	27	64
246	331b1cca79f04e3ba0c907bcf07224d1	38	31	38	40	46	68
247	33af29cb0deee7ee22f994f4a4d23a74	20	16	19	24	31	52
248	3498ca6576a3ec21cf28840ffd4db5e7	17	17	28	30	35	52
249	34a4c33ba5e4451c5796bb4476724d6a	15	15	26	15	16	72
250	3518cd0cebef50798acda338f243f16c	4	13	15	15	20	111
251	356ce264ae0867f60f34cd78a2f93ff0	39	35	39	41	53	63
252	35bc0e96dec5d36f55332ea649c373d6	14	13	21	22	15	78
253	364ff454dcf00420cff13a57bcb78467	314	152	362	314	341	385
254	38c940d037d653275b72c9de1b642727	103	60	116	119	161	174
255	3ec866180f9cac1bcb1d6037d2846567	319	313	317	319	413	417
256	3f037e9dd44b74b13d6791c6a2d69f10	29	25	28	35	44	53
257	427289af22c46174ecaf987d2178626d	20	16	19	24	30	537
258	454760fe8180c3c3bb062f8fc4aa1b7b	350	191	421	406	497	653
259	455ca63206d588da68c07d7bc2a6eeeb	39	35	38	45	58	143
260	458fe2439525b3f6b47ed4ba9d56f28e	15	15	23	15	16	59
261	45a02fb9272e3acb5c9a6c65bf41d768	17	16	27	32	35	52
262	45a943ce94b89de26ec923dd79b67c62	49	46	49	67	84	85
263	481d0baa98049379ab7713827393dc31	7	14	17	21	27	139
264	487bb61b3eeecb3988bb1d962b391470	21	19	20	21	35	44
265	49969f4484393afc1e1f41151512e1b4	19	16	18	19	29	43
266	4c78c0b15048a65721369ec3b076a4d3	26	24	32	34	44	56
267	4f46355e3b525340dba54aaef37513b9	60	59	65	71	89	154
268	51bba809f66c8d8df371f2c5ec690d68	14	11	30	30	37	486
269	51f516f91d06a0ea22b16a1499019784	17	17	27	17	22	63
270	528dded11385d5f6f0f2cd1aed767612	17	18	28	33	22	54
271	55127fe3361c858f792c1ed293979405	18	16	28	28	36	85
272	55889bba8c38037b64353664e71e4de2	19	15	18	22	29	42
273	55a410487b1b33320db189c7330d1d27	16	8	26	23	35	74
274	5835a68f0a6aca46219e2c3dd67bb08b	8	5	53	42	52	130
275	5a82854f4c17fdeb96d7573775d5c1f7	26	25	47	36	45	55
276	5d5c689616635c7f1f70e11f560cd7a9	15	14	27	27	34	47
277	61c3829b71be53cf531359f1179278f8	43	40	42	60	76	82
278	62e8fae3267ca477b5bcf6e20b08db5c	705	699	702	705	818	820
279	67e2781ab76e0fdf90e16feda6f9bb92	18	17	28	28	36	58
280	6bdbf23cef66b687d8770cdbb975152a	51	65	69	88	113	114
281	6db50873565946688adbc295b71df792	17	17	30	30	39	55
282	6f01828bff7489d75430922d882802ac	7	19	21	23	30	119
283	7058a6ff263e337c28d02555d4d5d840	193	150	266	194	243	323
284	706c0b48c89088fab58cb1eaa5cc8481	28	23	27	33	42	66
285	70da56d81aacfdd983032de8d153b134	19	15	18	23	29	41
286	71911c8703317d85550fb2c8434cba2e	11	5	20	16	29	118
287	719b1b9f691458af3b0da974649f42bf	8	7	24	22	28	486
288	71f0165f8f323fabeabb6e7899bd82d9	18	18	29	31	40	56
289	73e22cbf693132f18efd7de370b2c649	14	15	284	162	203	434
290	76f0a6e2e2b0041eb99fd38be1a10d30	18	17	29	25	20	52
291	7705b32ac794839852844bb99d494797	215	180	282	285	266	340
292	7731bca7a293366073a96bbeff46ef1e	26	22	25	26	43	59
293	78b3573a0b1c48e1ce7681590729b933	34	36	41	46	54	69
294	78facb6fed493a214931b38da717e0c7	17	17	28	32	22	64
295	797c5c00edd1b91cc97cc37ddc0efd4a	29	21	25	29	33	49
296	7b11921e962dd58a2a0d91c13f358e6f	21	18	31	35	43	82
297	80ea54e6b09a879a00496113146b9fe4	17	17	27	27	35	52
298	826c991fc57cb3ca593854c26b0e90d9	30	25	33	31	44	280
299	83c57db78a41143f9952f4dfa0be4e80	122	62	141	148	186	265
300	8416c4a84f495fe47f5cddece8afbb74	17	16	28	24	35	51

ID	MD5	Cuckoo	Habo	Padawan	Cuckoo++	X-Force	PMP
301	85c7e24b1c610e95a00e67de45306475	21	18	32	32	42	74
302	867ac455ee27fbd7872d3aefac729bf3	77	71	78	77	139	236
303	87113c55c5398c65c8aa7157b5b64f1a	8	5	53	32	41	77
304	88ecb91721cfa62e724317aa00293278	89	32	98	94	110	252
305	8932bd03aeaf81b1b6d6a7c97ea2da1a	8	7	24	15	20	168
306	893b1eddfcd390b26b8ddda3ac725fc6	51	47	50	55	90	97
307	8a4dabeef4e88749a6abe1d272003d15	72	67	73	77	125	126
308	8a82483ea34fd156010d9ea8a8234a49	43	39	42	61	75	81
309	8ab6624385a7504e1387683b04c5f97a	153	169	292	221	274	705
310	8d3ea75f160fdd9ff8efedab7e436851	51	48	50	71	89	95
311	8d68e286ebcaf2a3e76e7814bc4084fd	18	19	29	18	23	55
312	8f184c2e09d6e5c19e1edea50850c347	42	47	62	57	62	83
313	9188f0ff6070eb28b65aa1c396d89835	35	35	35	41	53	78
314	91f5d45b7a24d69e9d2d0b88870c8c40	27	23	26	27	40	98
315	9330d7d3114fc7bfa2ee8d05ad6882ec	17	17	27	31	29	54
316	937e25c1c059150dab0ec95a3a715262	21	25	31	21	27	75
317	961c824f208dbd57c2c489955830b195	30	25	29	30	45	56
318	969dd70e0dba7df04fc93548224ba8a2	17	17	28	33	39	55
319	984a0524e333060a337c5a6ceae06b42	18	18	28	32	41	64
320	98579b288581d02dcb2e6581f9be5a2f	68	63	69	70	72	74
321	9b0ee3bf1fa0a2b5e3d07c0b52dab1a6	43	40	42	43	75	81
322	9b6acc30fc9e224fea745906ed8f8889	23	21	33	35	44	58
323	9b7f5a1228fa66cbd35e75fb774fdc8e	153	203	246	161	199	655
324	9bb32c8115e3c643ee55dc41e754da73	7	10	17	12	25	139
325	a21e5260be784afbd01b93b20932ce8c	18	18	28	27	23	56
326	a27383a4644c8f25db5dfbd6496ab5d7	17	17	27	30	39	55
327	a27ee2b8f214dfbb5e15741751c09bf7	144	129	166	144	177	285
328	a4d4b5a8426822a6e26141f0a99781a4	14	14	25	25	32	45
329	a8c86a50e5613d2284c7e1a0f18e5bf2	15	15	25	22	33	73
330	a976deb51d295834b033609f9d5544ff	28	23	27	28	42	61
331	a9b6a5e7044ee975dbdbde90245f3938	18	19	29	18	23	56
332	aa5bebb84c2baae824782a85e2bde15a	58	52	60	58	77	258
333	aa646e4158bc48ecf4c745ef36664f1c	26	22	25	26	42	131
334	ab27fa9c2b797edacfbe961ae01372ad	28	24	27	28	43	63
335	ac6d049830db2f68ba01425be8b6d141	87	83	86	87	146	147
336	b03c32330edd83d10f23c941ce11412f	20	17	19	23	30	41
337	b04ab29c9a7a4fb99c1a8e60aebc5f38	17	17	28	26	22	54
338	b0c23492048f6cd5595cf847381fd5b2	20	17	19	20	30	544
339	b1598c6f6e9552b8c0776163793b529e	18	18	28	31	40	56
340	b27d6fa312b314d49a7e4ea7e85fc685	15	14	25	20	33	70
341	b2cd98a0b6f6ac9de92c92a702ee5f76	8	8	21	20	27	62
342	b4bc9b6f1c68981bad1cb40e8cf71e97	4	6	15	17	22	112
343	b58f043367e6057c9c79418d332e38c8	14	16	284	162	204	420
344	b8053ad0847830c698b0bdc35020f0d8	17	18	27	30	39	57
345	b85520dd2d64d6d05fe75b6112253fce	14	13	26	25	32	480
346	b931748458cfb2261cf7c14fb0441d95	24	20	23	27	34	45
347	bacee65f81615128345982051c4a605f	15	14	26	15	16	70
348	bc225bcbb80bef9c0b0d014305a9543d	55	54	54	72	92	100
349	bf2de60d4dddf43b4313668ad04ccafb	38	35	37	38	56	139
350	bfef696178596e2b801b396f8ec4c203	14	13	26	14	15	44

ID	MD5	Cuckoo	Habo	Padawan	Cuckoo++	X-Force	PMP
351	c39c03e276ac9bf64a502aa97f4187a9	21	18	35	34	27	89
352	c42554aef95702855ecc2c01f01d5cb9	72	65	67	99	124	125
353	c505029f2342e0452eed10d7705592fb	51	47	50	51	89	95
354	c61880e699640afbbba3e0ba7a8498b4	17	18	28	26	22	54
355	c8384a4b1951535448fe343374e38629	29	25	28	33	45	83
356	c87f1455ce2a5d3b68ce4bd4bb0f2ffb	17	18	26	29	22	54
357	c8d2fbac602fa261aa58276a2fd1c1d9	22	29	46	51	42	74
358	cba0943d3321347d28b293c14e2d352f	8	8	23	22	28	168
359	ccacb967524b88ec37f9779e826b89ea	24	20	25	27	34	45
360	cd3f835f1ef72f9dc48be1ea7f912dee	17	17	27	17	22	63
361	cd9db4354782ac9a26d9277d2d119ec6	133	133	168	167	138	449
362	cecd4988e023f5be02ae9fb8dbfd80c3	18	18	28	32	40	57
363	cff22e37378dbc280072c751cd13c612	75	71	74	103	130	131
364	d73face1dbd45383e74389a1bb3a2790	15	15	25	26	33	72
365	d766b045d130c0abc5d65be9254866d2	20	16	19	23	30	524
366	db5d478bdd8c50ee4425c3b7aa7a0342	19	12	20	22	29	263
367	dbd1c1eb767a458940a916a55e50783b	28	24	27	28	42	61
368	dc11905db6d7b885d0672836690b0789	17	17	27	22	39	55
369	dced35ba29cee86504064bf45c1fdd34	8	8	24	22	29	490
370	dd1e0191dbb0d9e6c30f6a17b968657e	39	35	39	39	53	63
371	de91ad771b54f73a924ac24a830c7bd9	17	16	28	17	18	53
372	e41f7965cba7e029c9c803274a928ef4	67	86	108	82	102	198
373	e4beb0caef120a317c73fc5640ef284b	15	14	25	26	33	71
374	e5c66d51421e6f90b8b7095d68f2c9fa	14	11	29	29	37	492
375	e7130e2ca5049be3acdb4fe01306f950	17	18	27	25	17	63
376	e8b597edd5d41bce904b6d417658c4bf	28	23	27	28	43	63
377	ead453a06315bfc702ad302821337fc2	20	33	49	45	65	70
378	eadfb2b01702d22f23e1af425f2613e9	17	18	24	17	22	65
379	ebd8790e97fb1403f72224429d6f89e4	43	39	42	43	76	83
380	ec52663c2e836fab94482c345aab9c5e	24	18	24	29	31	46
381	ed692adcc957fb54a24fe6e0c077c132	67	62	66	67	117	118
382	ee14c8b9fc8578f3218cd1da1ba46940	20	23	31	32	41	56
383	ee92d85933b024e8d82e03ed6acbaaf6	28	23	27	28	44	56
384	f0b820b96602eb7c63821df7cefe4ccd	38	34	37	38	56	135
385	f335f5857f2d30d0d811e1b732f0890a	15	14	25	15	16	69
386	f3c7855a2bc30b9d02baa8960a11f2ca	50	44	52	50	66	261
387	f3ff9415de6bab4f4c55d86e94ea1e85	319	315	317	327	412	416
388	f70d182ac7bb3d398ae47d38893dc1e2	201	188	203	205	258	268
389	f7e9e33108373f92527c3afd8a107aff	23	19	22	29	37	48
390	f8b42194ec19f3f5a7d7caedfb4188db	8	7	23	22	28	168
391	fa5c5264f4668f7a40f7576a27cfe78b	17	17	25	31	39	68
392	fb9c492cdaaf4a6be7032919c1f3a8df	20	16	19	23	30	550
393	fcb7184960449a616321c144090b3aa2	22	19	21	25	32	54
394	fcbfb234b912c84e052a4a393c516c78	263	35	283	263	285	298
395	fdb594009e2aa9f7a70f5e3c0b78cb86	18	18	28	32	40	56
396	fe681844084177d14a0a2e5d9ce9893e	77	88	89	94	104	228
397	fe742579bfbdd885a81fa16c57f7dcf7	15	14	26	30	33	73
398	febeaf981abcf790fb2f77d6c67ced7b	8	8	21	24	30	66
399	ff0c597903c66d6c5577c86cacde0baf	36	21	35	40	52	62
400	ff3ab2043c7a9c8d84ad785bb9301f83	15	14	25	26	15	69