

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

Wei You¹, Zhuo Zhang¹, Yonghwi Kwon², Yousra Aafer¹, Fei Peng¹, Yu Shi¹, Carson Harmon¹, Xiangyu Zhang¹

¹Department of Computer Science, Purdue University, Indiana, USA

²Department 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

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 explore 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., $SR(\text{rbx}) \leftarrow SM(\&\text{dests})$ such that $SR(\text{rbx}) = SM(\&\text{dests}) = \{0x7ffdf0000\}$, which

```

01 typedef struct(char ip[16]; long port;) Dest;
02 typedef struct(long act; Dest* dests[0];) Cmd;
03
04 int main(int argc, char *argv[]) {
05     Cmd *cmd = NULL;
06     int max = 0;
07
08     if (config_file_exists()) {
09         max = read_from_config_file();
10         cmd = malloc(sizeof(Cmd) + max*sizeof(Dest*));
11         for (int i = 0; i < max; i++)
12             cmd->dests[i] = malloc(sizeof(Dest));
13     }
14     ...
15     if (cnc_server_connectable()) {
16         scan_intranet_hosts(cmd, max);
17         cmd->act = get_action_from_cc_server();
18         switch (cmd->act) {
19             case 1: do_action_1(cmd->dest, max); break;
20             case 2: do_action_2(cmd->dest, max); break;
21             ...
22         }
23     }
24     ...
25 }

```

```

26 void scan_intranet_hosts(Cmd *cmd, int max) {
27     Dest **dests = cmd->dests;
28     for (int i = 0; i < max; i++) {
29         struct sockaddr_in *host = iterate_host();
30         inet_ntop(host->ip, dests[i]->ip);
31         dests[i]->port = ntohs(host->port);
32     }
33 }

```

```

α. mov rbx, [rbp - 0x10] // rbx = [rbp - 0x10] = [0x7ffdfbfed0] = 0x8
   /* Validate Memory Address: get_accessible(0x7ffdfbfed0) = true */
   /* Update Linear Set: SR(rbx) ← SM(&dests) = {0x7ffdfbfed0} */
β. mov ecx, [rbp - 0x14] // ecx = [rbp - 0x14] = [0x7ffdfbfec4] = 0x0
   /* Validate Memory Address: get_accessible(0x7ffdfbfec4) = true */
   /* Update Linear Set: SR(rcx) ← SM(&i) = {0x7ffdfbfec4} */
γ. lea rdx, [rbx + 8*rcx] // rdx = rbx + 8*rcx = 0x8
   /* Update Linear Set: SR(rdx) ← SR(rbx) = {0x7ffdfbfed0} */
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 = *(0x7ffdfbfed0) = 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 = *(0x7ffdfbfed0) = 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 |
| 0xffff | 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.

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

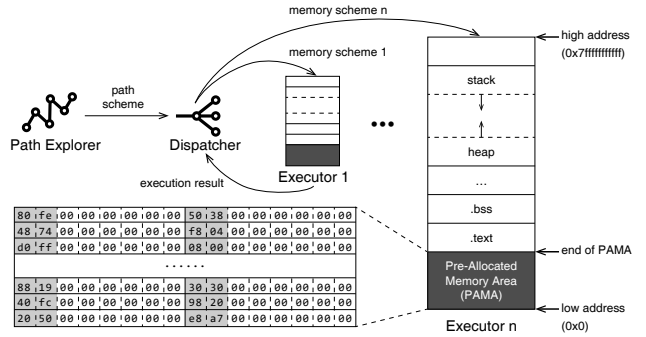


Fig. 3: Architecture of PMP.

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

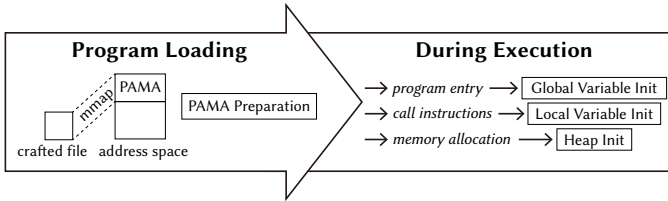


Fig. 4: Workflow of Memory-preplanning.

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 eight¹ 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 i th 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

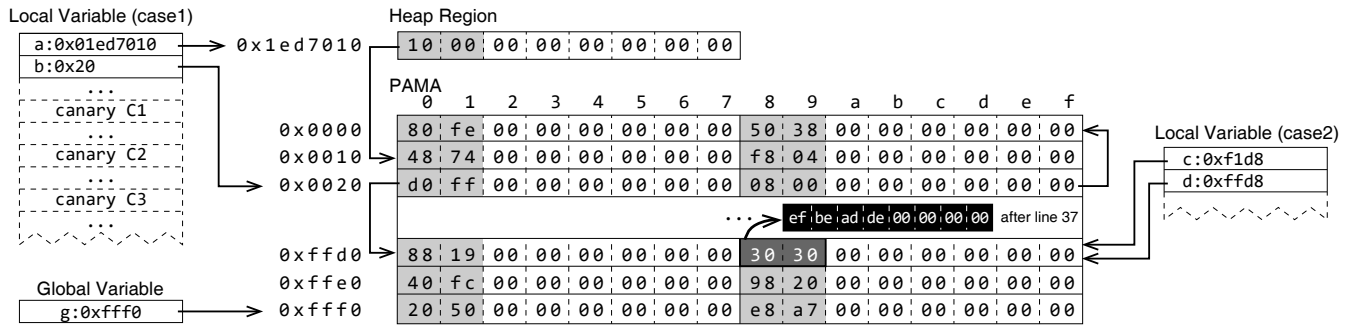
¹Eight is an empirical choice and works well in our evaluation. The number and the distribution of canaries are configurable.

```

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 `C1`, `C2`, and so on (see the stack frame in the top-left corner of Figure 5b), and then identifies, say, the region from `[C3, 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 `0xffff0` 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 dependencies and corrupted variable values. To facilitate discussion, we introduce the following definitions. Let \mathcal{P}_A be the set of all possible addresses within PAMA, and \mathcal{W}_A be its word-aligned subset. Assume the size of PAMA is S . Then, on a 64-bit architecture, we have equation (1).

$$S = |\mathcal{P}_A| = |\mathcal{W}_A| \times 8 \quad (1)$$

In addition, let \mathcal{FV} be a random subset of \mathcal{W}_A , 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 \mathcal{FV} . We define the ratio between the size of \mathcal{FV} and the size of \mathcal{W}_A as *diversity*, denoted as d . Then, we have equation (2).

$$|\mathcal{FV}| = |\mathcal{W}_A| \times d = \frac{d \cdot S}{8} \quad (2)$$

The initialization of PAMA can be formulated as a mapping $f : \mathcal{W}_A \mapsto \mathcal{FV}$, which assigns each word (with 8 bytes alignment) in PAMA (i.e., denoted by addresses in \mathcal{W}_A) with a random value selected from \mathcal{FV} . Intuitively, a more diverse \mathcal{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 \mathcal{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 `0xffff0` 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 \mathcal{FV} , α be an offset. The probability P_{err1} of $x + \alpha$ being out of the bound of PAMA is calculated by equation (3).

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

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

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 $S = 0 \times 400000$, $\alpha = 8$ and $d = 1$, $P_{err1} = 1.9073e-06$, 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 \mathcal{FV} . The probability P_{err2} of *coincidental address collision*, when x and y have the same value, is calculated by equation (4).

$$P_{err2} = P(x = y \mid x \in \text{FV}, y \in \text{FV}) = \frac{8}{d \cdot S} \quad (4)$$

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

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

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

Proof is elided due to space limitations. With a setting of $\beta = 0 \times 1000$, $\gamma = 0 \times 1000$, 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 $ga \in GA$, $ha \in HA$, 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 FV' be the set of the adjusted filling values. Then we have $\text{FV}' = \{x - base \mid x \in \text{FV}\}$.

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 0xffff0 set by PMP. If its pointer field `f4` is not word-aligned, its value will be loaded from 0xffff1, 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., $S = 0 \times 400000$), 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 (83%) | 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 (94%) | 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.perlbnk | 1,189 | 10,318 (8.7/s) | 508 (5%) | 118,135 (88%) | 11,600 (92%) | 692 (97%) | 66,726 | 28,394 (43%) | 4,001 (6%) | 176,096 | 10,400 (0.06/s) | 502 (4%) | 119,467 (89%) | 11,676 (97%) | 696 (91%) | 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 | 20,651 (54%) | 3,059 (8%) | 103,458 | 7,461 (0.07/s) | 298 (4%) | 49,920 (54%) | 4,521 (50%) | 401 (88%) | 38,784 | 18,228 (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 (97%) | 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*.

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

```

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*.

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.

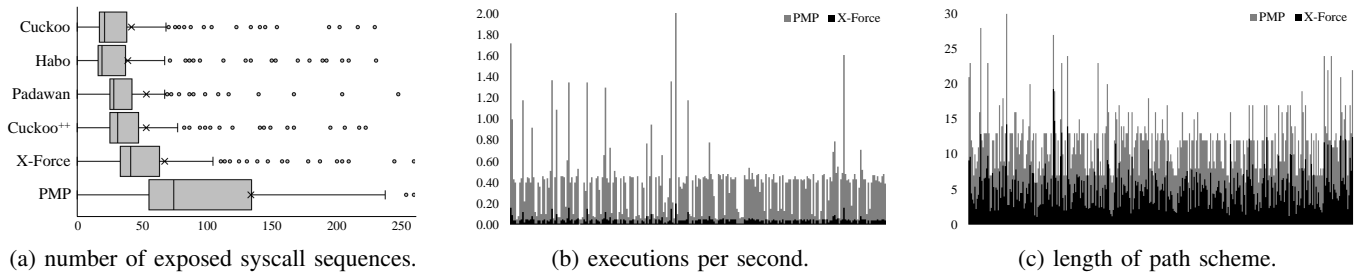


Fig. 8: Overall result of malware analysis.

TABLE II: Experiment with *mcg*.

| 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

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 |

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.

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 mmmaps 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/ReVil12")) 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 through 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

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


```

01 int main(int argc, char **argv) {
02     struct sysinfo info;
03     sysinfo(&info);
04     if (info.uptime < 128) exit(0);
05     DIR *dir = opendir("/etc/init.d");
06     while (struct dirent *ent = readdir(dir)) {
07         char name = ent->d_name;
08         if (name[0] == 'S' && is_num(name[1]))
09             add_to_init_script("/etc/init.d/S99");
10     }
11 }

```

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

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

```

01 char *data = read_file("/sys/class/dmi/id/product_name");
02 if (contains(data, "VirtualBox", "VMware"))
03     remove_self_and_exit();
04 while (1) {
05     char *ip = select_intranet_ip(ip_list);
06     char *vuln = select_known_vuln(vuln_list);
07     if (connect_and_check(ip, vuln)) {
08         send_info_to_server(ip, vuln);
09         send_payload(ip, vuln);
10     }
11 }

```

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

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

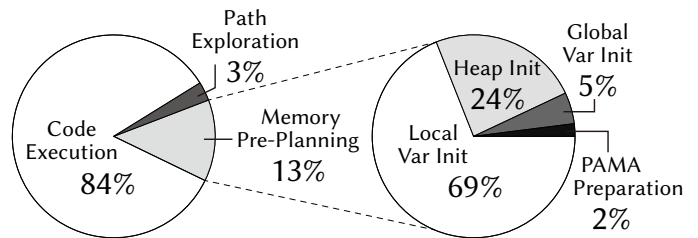
- [1] Clickless powerpoint malware installs when users hover over a link. <https://blog.barkly.com/powerpoint-malware-installs-when-users-hover-over-a-link>.
- [2] Cuckoo. <https://cuckoosandbox.org/>.
- [3] Cybersecurity statistics. <https://blog.alertlogic.com/10-must-know-2018-cybersecurity-statistics/>.
- [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_\(malware\)](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. Omnium-pack: 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 web-view 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 | Habo | Padawan | Cuckoo++ | X-Force | PMP |
|-----|----------------------------------|------|---------|----------|---------|-----|
| 051 | 3decf1b4e5e821c159e051a04bf0452 | 7 | 20 | 19 | 27 | 27 |
| 052 | 3c21a608b64341e97a7861fa08024ec2 | 20 | 18 | 32 | 49 | 61 |
| 053 | 3f193286767c269b786e117c43807f7b | 17 | 18 | 27 | 32 | 40 |
| 054 | 3f0857173602653861bd4054749b395 | 14 | 14 | 27 | 27 | 34 |
| 055 | 4084544a9024e144b4f2453c11dc4003 | 18 | 18 | 29 | 28 | 41 |
| 056 | 4087376e772170248eb2f0665a26796 | 22 | 19 | 22 | 25 | 33 |
| 057 | 424f94d07b45eab1bd32494cdeb467b | 22 | 45 | 40 | 35 | 22 |
| 058 | 46eaf3f07c2e590bb2847aaeb41de4 | 40 | 37 | 39 | 44 | 62 |
| 059 | 483b322b2483527798f5239df5c6fc | 30 | 7 | 26 | 30 | 43 |
| 060 | 49c17897f650c77db316234cfc5eb | 17 | 18 | 27 | 25 | 40 |
| 061 | 4b1e9e8cc1919983935092904436edc3 | 60 | 61 | 59 | 60 | 77 |
| 062 | 4c593af1a625873681c62ca4f49e31c3 | 21 | 19 | 31 | 35 | 43 |
| 063 | 4f5d0ed102de7c171d1df4989c4cd40 | 15 | 16 | 25 | 25 | 36 |
| 064 | 4faa269b7ce44bce5f5746ea37c38f | 25 | 16 | 21 | 25 | 37 |
| 065 | 502a90ed7a851b01b340ade4822c4de0 | 28 | 22 | 27 | 33 | 41 |
| 066 | 5242874da3d6d8c59ebc249476ed8181 | 27 | 23 | 26 | 32 | 42 |
| 067 | 53ad943fe07bc315d9908c668f305a08 | 24 | 22 | 35 | 40 | 46 |
| 068 | 54b0f1404a40e5713377ad4a8f143ad | 24 | 17 | 25 | 26 | 34 |
| 069 | 559169e48167dcbaf065d6a122a289d | 20 | 43 | 38 | 33 | 40 |
| 070 | 55e0a8737b091da7bd47060b75b2e119 | 60 | 61 | 62 | 60 | 101 |
| 071 | 56eb1e4e788e63325bb551da187e609 | 31 | 27 | 59 | 64 | 70 |
| 072 | 57b1f91b59aad9a1c5669404bd4d46a | 27 | 22 | 27 | 27 | 49 |
| 073 | 57b4d2108051d8e43d7b35777b76d40 | 15 | 15 | 26 | 27 | 34 |
| 074 | 582f47e9e75b0ba8eaf5a9ccbd552 | 24 | 22 | 34 | 36 | 46 |
| 075 | 58af33bfaf681eb637b59a20ba4ea0c3 | 26 | 53 | 48 | 51 | 65 |
| 076 | 5a6fd6344ffc6037dc192b6c3f456e87 | 55 | 32 | 58 | 58 | 76 |
| 077 | 5b36aebd504673123e104e21529b638 | 21 | 19 | 31 | 34 | 43 |
| 078 | 5c1dd20f744ac8230686441196171c | 140 | 149 | 87 | 140 | 157 |
| 079 | 5c4709a37370b964e97518e435de9 | 17 | 18 | 27 | 27 | 39 |
| 080 | 5cf611021b8012131577e85bf81a82f | 22 | 45 | 40 | 47 | 60 |
| 081 | 5d6aa67ce3427036f735925d359c3049 | 43 | 40 | 42 | 43 | 77 |
| 082 | 5e890eb3f6c8a8168d078fdded09996 | 29 | 25 | 28 | 29 | 44 |
| 083 | 5f13326e2c90b70593b645540f25213f | 17 | 18 | 28 | 32 | 40 |
| 084 | 5fb565ee5336c0b30451a0a02303608 | 11 | 5 | 20 | 20 | 29 |
| 085 | 5fd2e44f2f0cc7014821b7b78a00b1 | 36 | 7 | 36 | 37 | 76 |
| 086 | 5fea85c62d1117a7931df0f8b62dd3 | 21 | 24 | 31 | 33 | 43 |
| 087 | 6025e14e0487c35e8a049885f035b97b | 15 | 15 | 23 | 25 | 32 |
| 088 | 6139657db08c3e9d5d2399259e8caad0 | 28 | 24 | 27 | 33 | 43 |
| 089 | 62c2d296060d14061f5c54f31662dae9 | 31 | 27 | 57 | 61 | 88 |
| 090 | 6355f0eaf619090eb0baed57016bb6c | 19 | 20 | 19 | 19 | 31 |
| 091 | 664378d101610552d17e97cc06ade139 | 20 | 43 | 48 | 39 | 48 |
| 092 | 60dbd9599779c3a4899a0ee9f2eeeb03 | 28 | 24 | 28 | 35 | 45 |
| 093 | 6bd1557eac7093ee9e5807385db0b05 | 15 | 15 | 26 | 22 | 34 |
| 094 | 705df7bc13a3c1bbfc79735455fda68 | 24 | 21 | 25 | 28 | 34 |
| 095 | 71d6a0b0940e3ff974833d3d729688d | 29 | 29 | 29 | 45 | 173 |
| 096 | 717dfa046833dae608b6f1a274a47938 | 8 | 7 | 23 | 23 | 29 |
| 097 | 72afcb455fa4bc1e5f16ee67c6f915 | 362 | 362 | 291 | 362 | 423 |
| 098 | 74124dae8fddb903bec57d5bc31246b | 36 | 40 | 36 | 38 | 40 |
| 099 | 74f0ec75b66ced0be2e4e5455e90a5 | 41 | 7 | 40 | 41 | 44 |
| 100 | 75e04ad828359d2d25718430bc5f3dd3 | 14 | 14 | 24 | 26 | 33 |

| ID | MD5 | Habo | Padawan | Cuckoo++ | X-Force | PMP |
|-----|------------------------------------|------|---------|----------|---------|-----|
| 001 | 00056adcf6982498c184f429d7af61d4 | 29 | 28 | 31 | 42 | 92 |
| 002 | 005449f26b000333c8bac5cb15e2c60f6b | 15 | 15 | 27 | 34 | 74 |
| 003 | 0191642afcabbbcb2e9449822ca10d37 | 70 | 65 | 69 | 98 | 126 |
| 004 | 03cfe768a8b4fbb0b6f0dfe986389dc | 27 | 29 | 27 | 32 | 216 |
| 005 | 045136430edac124e134bf2a32a4a60 | 15 | 15 | 15 | 16 | 33 |
| 006 | 057857302490521bd52425a141bbdbfb | 14 | 14 | 27 | 34 | 591 |
| 007 | 0686a7459132174f821c86c33cfba8a | 47 | 53 | 47 | 49 | 90 |
| 008 | 08ab6111b6b3d574036cf10fe787063 | 14 | 14 | 25 | 27 | 33 |
| 009 | 09e4b26f6b499a81453766c17226106 | 22 | 32 | 27 | 37 | 46 |
| 010 | 0b85584df6a3ca84d5f6931570e02ae | 48 | 53 | 48 | 55 | 69 |
| 011 | 0be2cbb5fe3e051355a50c07885464bf | 15 | 15 | 29 | 29 | 17 |
| 012 | 0c0d2ed33316dc3a29a2785007dcb50 | 7 | 10 | 17 | 19 | 25 |
| 013 | 0c1aa91e8cae4352eb16d931f7c0da2b | 15 | 15 | 23 | 23 | 74 |
| 014 | 0fe8985c56da5a821f9b135aa3dbd4 | 22 | 35 | 30 | 35 | 44 |
| 015 | 0d186ccf5829dd5bf1dc2aff944fc2f6 | 23 | 21 | 34 | 37 | 45 |
| 016 | 10c47191922eeefcae39bf5bce540bd44 | 15 | 16 | 23 | 27 | 35 |
| 017 | 10f5bea257a926638666c9955087bb | 20 | 17 | 19 | 25 | 31 |
| 018 | 113c079464639ba412876b42c1d96ac7 | 24 | 22 | 35 | 35 | 46 |
| 019 | 11e489ddea858030b2377ac184994439 | 48 | 54 | 47 | 55 | 69 |
| 020 | 1226c436e5e830c9b9f8a043fa4f913b | 43 | 40 | 42 | 59 | 76 |
| 021 | 1321bd12e164aa78b7e39ef7bce8a62 | 20 | 18 | 31 | 27 | 42 |
| 022 | 132397af7e931be40528d44634a1582 | 36 | 40 | 36 | 50 | 63 |
| 023 | 137c1520b37dca3e5072be7995c96fc | 14 | 14 | 24 | 26 | 33 |
| 024 | 13f2bb2af16f513ba435a2c6c8f85cbe | 40 | 41 | 42 | 40 | 58 |
| 025 | 175973131f49959e2b1a75a703562d4bf | 17 | 18 | 27 | 26 | 40 |
| 026 | 179c76488b607147973c2f6cbe0e530 | 21 | 25 | 31 | 34 | 43 |
| 027 | 199e8ffe248a35e99e1726ff796d9398 | 14 | 20 | 14 | 18 | 32 |
| 028 | 1a7e8ddc317806db053c47e12991c33 | 15 | 15 | 25 | 26 | 34 |
| 029 | 1b5054939ee601d89fdaa44c109943cf | 29 | 22 | 28 | 29 | 42 |
| 030 | 1b74c8a749948421b12190486cc631cf | 17 | 18 | 27 | 30 | 40 |
| 031 | 1e19b857a5f5a9680555f9a623a88e99 | 12 | 17 | 12 | 12 | 283 |
| 032 | 2077166b21e9717f076ca897e5bf94 | 14 | 14 | 14 | 15 | 44 |
| 033 | 210e4243c8ed874999c7caa4076d433 | 22 | 45 | 41 | 40 | 60 |
| 034 | 22dcd1ba876721727cca37c21d31655 | 5 | 8 | 5 | 7 | 17 |
| 035 | 23c42760532270113be57b97346cddf0 | 20 | 18 | 30 | 30 | 42 |
| 036 | 24bf1279bc8ffe08c380675cbb81b94a | 17 | 17 | 27 | 30 | 39 |
| 037 | 25c364af98025dca816ac10e8283af | 17 | 18 | 28 | 32 | 40 |
| 038 | 28255eb4c29e0420572126d8bc0e481 | 20 | 18 | 30 | 30 | 42 |
| 039 | 28e86843a9462113eb26ae1024db08 | 17 | 17 | 28 | 29 | 36 |
| 040 | 28fcd854ceadd32abf0946e0692e9ac4 | 21 | 19 | 32 | 33 | 42 |
| 041 | 2a4284994083eb88456ed361d7c381 | 22 | 45 | 40 | 41 | 53 |
| 042 | 2d66f629e00042d8662b38483c733bb | 20 | 30 | 25 | 24 | 43 |
| 043 | 2e64529eac407d4e47b70b72082490c | 20 | 18 | 30 | 33 | 42 |
| 044 | 31c55141129151ee4728a40613b93cca | 21 | 17 | 21 | 21 | 31 |
| 045 | 3544c1e682497dc5d8bf6898f17fcf | 17 | 18 | 28 | 32 | 40 |
| 046 | 36263491d776cd4b93b97ead05a8656a | 36 | 40 | 36 | 36 | 63 |
| 047 | 36a32f5a8a058df437fa67ecce06cf | 39 | 36 | 38 | 40 | 58 |
| 048 | 39a46a0cd60393c5571b720c915db30d | 48 | 54 | 47 | 54 | 69 |
| 049 | 3a0f68a257cfa2d11292cb6420ed884a | 18 | 19 | 28 | 26 | 41 |
| 050 | 3b0d923cf1792151e6540ca38b3d6d19 | 20 | 17 | 19 | 20 | 32 |

| ID | MDS | Cuckoo | Habo | Padawan | Cuckoo++ | X-Force | PMP |
|-----|------------------------------------|--------|------|---------|----------|---------|-----|
| 151 | c2764861eac7f3cda2227bfe6b7f07d | 8 | 8 | 7 | 10 | 13 | 129 |
| 152 | c2a5b75e7273b3b4dbf0a02343ea35f2 | 28 | 24 | 27 | 29 | 43 | 64 |
| 153 | c32a5d9b0c78b335af5197d3831966a9 | 41 | 42 | 41 | 41 | 50 | 61 |
| 154 | c36625389cb4739518742dc4298536fb | 54 | 32 | 58 | 60 | 75 | 120 |
| 155 | c38f08b904d5e1c7c798e840f1d8f1ee | 82 | 84 | 87 | 86 | 110 | 137 |
| 156 | c533142180337f002f5e2a0e2bf9e099 | 14 | 14 | 27 | 27 | 34 | 587 |
| 157 | c63cef04d931d18171d0c40621855e9 | 15 | 15 | 26 | 31 | 34 | 79 |
| 158 | c64919c972336dced4e97140c1533b274 | 14 | 7 | 35 | 31 | 46 | 327 |
| 159 | c80b8f2a2d6a9e1500bfa52f864e46d | 17 | 18 | 28 | 31 | 40 | 55 |
| 160 | c83b5e8b47824392082e284240b218b4 | 17 | 18 | 27 | 31 | 40 | 55 |
| 161 | c8c1f2das11fb0f0eaa60e11a81236e94c | 29 | 22 | 28 | 29 | 41 | 91 |
| 162 | c97acdf1fac05a080a7825f5647d4244a | 17 | 18 | 28 | 32 | 40 | 55 |
| 163 | cb0477445f9c5f1a3b6689bbfb941e | 70 | 66 | 70 | 99 | 125 | 127 |
| 164 | cb3d93f65c64e48e181274a49a748ce7 | 29 | 25 | 28 | 36 | 45 | 116 |
| 165 | cc29a224c327412e0db713ce5c4f4e00 | 6 | 8 | 6 | 14 | 18 | 35 |
| 166 | cd60f742f2f1f98b34a264c5f3e55ae42 | 14 | 14 | 29 | 25 | 34 | 34 |
| 167 | cfcd5153e739406baa7b3354dd5b28e04 | 17 | 18 | 28 | 31 | 40 | 55 |
| 168 | d04c492a5b78516a7a36cc2e1e8b5f21 | 70 | 66 | 69 | 99 | 125 | 127 |
| 169 | d0874ba3c4fbdf714fct2c0a117cc8e2 | 38 | 35 | 37 | 38 | 58 | 135 |
| 170 | d0b9d58f3a454ad6df2e4d0558581e5 | 6 | 9 | 6 | 14 | 18 | 33 |
| 171 | d1a19e834de3447ecfd8af04c0e6be4 | 14 | 14 | 28 | 23 | 34 | 587 |
| 172 | d21fb7ed52ba1329424f034c1f528d2f | 3 | 5 | 3 | 10 | 13 | 269 |
| 173 | d2cd482ba82e592c1dc5dcd7db79c70 | 15 | 25 | 25 | 24 | 34 | 74 |
| 174 | d3a894f6052eece1ca87b69c619ea0cb | 31 | 27 | 30 | 31 | 49 | 122 |
| 175 | d493af745de315c6989355a49d21b2a3 | 20 | 18 | 31 | 42 | 42 | 56 |
| 176 | d721e7eb5d63ea8f85540748924f301d | 42 | 42 | 43 | 42 | 48 | 52 |
| 177 | d7d73062d2efef11b6ba3bdaf5e4e18 | 17 | 17 | 28 | 33 | 36 | 55 |
| 178 | d979d2dce979788c0e9cc72b445617 | 27 | 32 | 48 | 60 | 74 | 74 |
| 179 | dae9f1e16b6fee713f53182c2d4de10 | 17 | 6 | 28 | 31 | 40 | 40 |
| 180 | db16765a02afbe75ae569c5901744c19 | 346 | 358 | 460 | 465 | 522 | 712 |
| 181 | dc4db38f6d3c1e751dcb06bae072ba9c | 15 | 15 | 26 | 29 | 34 | 75 |
| 182 | dd7f74445d61c8d80335b15d432c27b | 13 | 7 | 21 | 26 | 33 | 110 |
| 183 | de2e41048e3a54ac1e6bbae91ae999ab | 20 | 43 | 39 | 42 | 30 | 59 |
| 184 | de5798b69df92163cdd25f362565c521 | 27 | 32 | 49 | 44 | 74 | 74 |
| 185 | df09a1a31fadaad518a6760c3cfbdc17 | 28 | 46 | 42 | 51 | 64 | 170 |
| 186 | e37ff9a31c89b929e963333aa72f96 | 40 | 37 | 40 | 40 | 60 | 135 |
| 187 | e3d802cd1de02c74f198189aba33052 | 29 | 22 | 28 | 29 | 42 | 91 |
| 188 | eeffa02a63c951e4e8a131e43d9fca6a | 15 | 15 | 25 | 20 | 34 | 76 |
| 189 | ee3de1355a2056a7eb5e799b5e989d0b | 24 | 47 | 44 | 49 | 63 | 92 |
| 190 | ee673fed52823da1ebae7019e042382 | 21 | 19 | 31 | 28 | 42 | 82 |
| 191 | ed62ce1a4d062e0b9d6d79ca4e3572b6 | 18 | 18 | 28 | 33 | 37 | 59 |
| 192 | ee11c23377f5363193b626da566b9f5c | 31 | 27 | 57 | 43 | 55 | 87 |
| 193 | f27751a1292f2521cc55f190f15bd30b | 14 | 22 | 284 | 162 | 203 | 423 |
| 194 | f2b00b27e6e8110d3c27525ecd9af120 | 47 | 53 | 48 | 53 | 69 | 92 |
| 195 | f3e8a50f0c1c3a510f882d0fbd121960 | 14 | 14 | 24 | 24 | 33 | 33 |
| 196 | f8f62b7f01c3a2610a9d33b6c5f51c | 17 | 16 | 27 | 27 | 36 | 36 |
| 197 | fa68eb454b37401bb0476248a3eb4845 | 20 | 17 | 19 | 24 | 31 | 65 |
| 198 | fa7a3c2574284c7fda9f6ac67311eda | 24 | 14 | 20 | 27 | 34 | 159 |
| 199 | fd75a87293ca3213f5c033f64feef0f | 18 | 17 | 29 | 30 | 37 | 58 |
| 200 | ff02a16427e32005262203501a8e9b4f | 30 | 26 | 30 | 35 | 44 | 55 |

| ID | MDS | Cuckoo | Habo | Padawan | Cuckoo++ | X-Force | PMP |
|-----|------------------------------------|--------|------|---------|----------|---------|-----|
| 101 | 770756fdac23e4f3c0a17f26bc22b6 | 17 | 17 | 28 | 28 | 35 | 56 |
| 102 | 7dad01126f01992d24f00f86d08d042e | 37 | 18 | 28 | 24 | 40 | 55 |
| 103 | 81b6ec216e10c17104706336c21a479a | 39 | 36 | 39 | 48 | 59 | 157 |
| 104 | 81ea379c237724249c137f683ef21e9a | 6 | 9 | 6 | 6 | 19 | 35 |
| 105 | 850177156d5a010254bb5746664a3c7 | 15 | 15 | 26 | 27 | 34 | 34 |
| 106 | 862ef4928c8edf150e22e08bb14c61 | 17 | 18 | 27 | 25 | 40 | 56 |
| 107 | 898d6eab3142e60075283599b935e9e | 48 | 54 | 47 | 48 | 69 | 88 |
| 108 | 8hd0c5f6987218a95dc56677c40f880 | 17 | 17 | 27 | 29 | 36 | 57 |
| 109 | 8bf6de4ef1067ca119d4d71a66a84e06e | 8 | 7 | 25 | 23 | 29 | 29 |
| 110 | 8c5c16e2d737f0d0c36b2c1252ddd75 | 37 | 40 | 38 | 50 | 63 | 266 |
| 111 | 8dbd0738910e3f4590bea87a3c1ac538 | 27 | 32 | 48 | 51 | 74 | 74 |
| 112 | 8df9ec7cd1de78957ea800f063d66051 | 39 | 36 | 39 | 39 | 58 | 132 |
| 113 | 8f194847387186899cc8d9f9ca903e07 | 103 | 112 | 139 | 143 | 147 | 190 |
| 114 | 901cbff40784ee40518fda6471e70baa | 15 | 14 | 26 | 26 | 33 | 70 |
| 115 | 912bcas947944fdcd09e9620d7aab4a | 15 | 15 | 25 | 26 | 34 | 83 |
| 116 | 9353a060cc5f82f6ce8a0105dfac48f | 15 | 14 | 26 | 24 | 32 | 34 |
| 117 | 9361ad45b4bf3041759bd4f72920d2 | 14 | 14 | 28 | 33 | 34 | 587 |
| 118 | 93c2f1ca9949435cffe81572d3d21d5e | 15 | 15 | 25 | 27 | 34 | 34 |
| 119 | 942ea0c4c7294d4878e5b8998981228 | 48 | 54 | 48 | 54 | 69 | 92 |
| 120 | 96804156396bc25d49c4e4af0584569 | 47 | 53 | 48 | 47 | 59 | 83 |
| 121 | 96dc2982978ca899b4a97f73e7f466 | 15 | 15 | 26 | 27 | 34 | 76 |
| 122 | 97ba4828e2856d8e15c1c916585e | 17 | 18 | 28 | 25 | 42 | 57 |
| 123 | 994136a318399900f73d083bf42a330 | 97 | 94 | 109 | 109 | 138 | 142 |
| 124 | 9a2b507212c19a9d9c95168745e793ea | 39 | 36 | 38 | 39 | 59 | 145 |
| 125 | a25470a5305f5c7c80b68810e132b2 | 43 | 40 | 42 | 46 | 76 | 83 |
| 126 | a27896388f0d4d493e7d786e4eaaab | 14 | 14 | 23 | 21 | 15 | 95 |
| 127 | a3ab4d03e3b16f0ed144923db29daec | 20 | 30 | 25 | 23 | 42 | 53 |
| 128 | a4404be674d1f144ea86a7838f357c26 | 47 | 43 | 46 | 55 | 69 | 82 |
| 129 | a4944230462083019d13a8f61b476f53 | 14 | 15 | 24 | 25 | 35 | 54 |
| 130 | a4eeef761f4c90b8065800d4cad391df | 29 | 40 | 59 | 56 | 78 | 303 |
| 131 | a58f8b3be409874271fa04709012b5ad | 19 | 17 | 29 | 32 | 41 | 55 |
| 132 | a62f2bca5c0a5d239c6a3732a2f424ab | 228 | 229 | 359 | 355 | 408 | 621 |
| 133 | a6617c5cb59135e057994984d264564c7 | 75 | 71 | 72 | 75 | 95 | 129 |
| 134 | a664d72a34b8653f0a6e04c9e866d4c | 17 | 18 | 28 | 33 | 39 | 172 |
| 135 | a71079102c6f7053a9402f72cc979825 | 22 | 18 | 21 | 22 | 22 | 33 |
| 136 | a8cd638e13b1848f347c724e9386ea8 | 15 | 15 | 23 | 25 | 33 | 79 |
| 137 | a8f78241bd7b7cad50e054bcb4df0a1b | 27 | 32 | 48 | 47 | 74 | 74 |
| 138 | a961ce018d771932b70a9f9eb8b7484 | 347 | 353 | 359 | 415 | 523 | 713 |
| 139 | ab2f936e95daf91789caaa802ec4948cf | 22 | 19 | 21 | 26 | 34 | 65 |
| 140 | ab40bca438fbf809b5786d52b38ea318 | 39 | 36 | 39 | 39 | 59 | 144 |
| 141 | abbb05240c9484c5a30b7f348e225b31 | 18 | 19 | 28 | 31 | 40 | 41 |
| 142 | ac2e9ce2b3cedf07045024d60f9ba4e53e | 27 | 32 | 48 | 60 | 75 | 75 |
| 143 | ad76e4b7470d1936380b2b5375410b4 | 40 | 37 | 39 | 49 | 62 | 148 |
| 144 | acc2d8a6c35aac5b01b049f1879aa1 | 20 | 30 | 25 | 23 | 42 | 49 |
| 145 | b4088daeb311c24d8f9a2d0b5ec223bc9 | 21 | 17 | 20 | 24 | 31 | 55 |
| 146 | b754622e816b2281402b867f5fa9ccf | 26 | 22 | 25 | 26 | 42 | 337 |
| 147 | b8f6cd7360dd2411feb8d6c77b775 | 15 | 14 | 25 | 22 | 34 | 34 |
| 148 | b91fd8175009c377ca9c799e987c74 | 27 | 32 | 48 | 60 | 74 | 74 |
| 149 | be04b91301e51c3424be7841b13fd05 | 15 | 15 | 26 | 26 | 34 | 77 |
| 150 | bf8287805atdfc72ca6b7c6c76d5b04a | 347 | 355 | 352 | 351 | 521 | 716 |

| ID | MDS5 | Habo | Padawan | Cuckoo++ | X-Force | PMP |
|-----|-----------------------------------|------|---------|----------|---------|-----|
| 251 | 35cc2e64ae0867f60f34cd78a2d93f0 | 39 | 39 | 41 | 53 | 63 |
| 252 | 35bc0e96d6c5436f5532e649c373d6 | 14 | 13 | 21 | 15 | 78 |
| 253 | 364ff454cd00420c1f3a57bcb78467 | 314 | 152 | 362 | 314 | 385 |
| 254 | 38e940037d653275b72e9de1b642727 | 103 | 60 | 116 | 119 | 174 |
| 255 | 3cc86618099cacc1bc1d6037d2846567 | 319 | 313 | 317 | 319 | 417 |
| 256 | 3037e9ad44b74b13d6791c6a2d69f10 | 29 | 25 | 28 | 35 | 53 |
| 257 | 427289af22c44714eca1987ad2178626d | 20 | 16 | 19 | 24 | 537 |
| 258 | 454760e8180c3c3db062f81c4aa1b7b | 350 | 191 | 421 | 406 | 653 |
| 259 | 455ca6320664588da68c07f7bc2afceeb | 39 | 35 | 38 | 45 | 143 |
| 260 | 458fe2439525b3f16b47ed4ba9d56f28e | 15 | 15 | 23 | 15 | 59 |
| 261 | 45a02f97272e3bc5e9a6c65bf4d768 | 17 | 16 | 27 | 32 | 52 |
| 262 | 45a943ce94f0899e26cc923dd79b67c62 | 49 | 46 | 49 | 67 | 84 |
| 263 | 48140ba90409379ab7713827393dc31 | 7 | 14 | 17 | 27 | 139 |
| 264 | 487bb61b3eeecb3988b1d962b591470 | 21 | 19 | 20 | 21 | 35 |
| 265 | 49969f4484393af1e1f41151512e1b4 | 19 | 16 | 18 | 19 | 43 |
| 266 | 4e78c0b15048a65721369cc3b076ad43 | 26 | 24 | 32 | 34 | 56 |
| 267 | 4f46355c3b525340dba54aaef37513b99 | 60 | 59 | 65 | 71 | 154 |
| 268 | 51ba809166c8d8f371f2c5ec69d68 | 14 | 11 | 30 | 30 | 486 |
| 269 | 51f51691d06a0ea22b16a1499019784 | 17 | 17 | 27 | 27 | 63 |
| 270 | 528dded11385d5f6f0f2cd1aed767612 | 17 | 18 | 28 | 33 | 54 |
| 271 | 55127fe3361c8587f92e1ed293979405 | 18 | 16 | 28 | 28 | 85 |
| 272 | 55889bba8c38037b64353664e71e4dc2 | 19 | 15 | 18 | 22 | 42 |
| 273 | 55a410487b1b33320db189c7330d1d27 | 16 | 16 | 26 | 23 | 74 |
| 274 | 5835a68f0ba6ca46219e2c3dd67b08b | 8 | 5 | 53 | 42 | 130 |
| 275 | 5a28254f4c71fde9e9d7573775d5c1f7 | 26 | 25 | 47 | 36 | 55 |
| 276 | 5d5c689616635e71f70e11f566c47a9 | 15 | 14 | 27 | 27 | 47 |
| 277 | 61c3829b71bc53c5131359f17927818 | 43 | 40 | 42 | 42 | 82 |
| 278 | 62e8fae3267ca077b5bcfca2b08db5c | 705 | 699 | 702 | 705 | 820 |
| 279 | 67e2781ab76e0fd090e16feda6f9bb92 | 18 | 17 | 28 | 28 | 58 |
| 280 | 6bd1b23ce6f6b687d8770cbb975152a | 51 | 65 | 69 | 88 | 114 |
| 281 | 6db50873565946688adbc295b71df792 | 17 | 17 | 30 | 30 | 55 |
| 282 | 6f01828bf7489d75430922d882802ac | 7 | 19 | 21 | 23 | 119 |
| 283 | 705846ff263c37c28d02555d4d5d840 | 193 | 150 | 266 | 194 | 323 |
| 284 | 706c0b48c8908f8ab58cb1eaa5cc8481 | 28 | 23 | 27 | 33 | 66 |
| 285 | 70da56d81aacfd983032e8d1153b134 | 19 | 15 | 18 | 23 | 41 |
| 286 | 71911c8703317d85550fb2c843cba2c | 11 | 5 | 20 | 16 | 118 |
| 287 | 719b1b9f691458af3b0da974649f42bf | 8 | 7 | 24 | 22 | 486 |
| 288 | 71f0165f8f323fabcaab6c78999d82d9 | 18 | 18 | 29 | 31 | 56 |
| 289 | 73e22cbf6931321f18edf7de370b2c649 | 14 | 15 | 284 | 162 | 434 |
| 290 | 76f06e2c2b0041eb99f1cc37dcbef44 | 18 | 17 | 29 | 25 | 52 |
| 291 | 7705b32ac794839852844bb99494797 | 215 | 180 | 282 | 285 | 340 |
| 292 | 7731bca7a293366073a96bbbf46effe | 26 | 22 | 25 | 26 | 59 |
| 293 | 78b3573a0b1c48e1cc7681590729b933 | 34 | 36 | 41 | 46 | 69 |
| 294 | 78facb6fe4d93a214931b38d871e0c7 | 17 | 17 | 28 | 32 | 64 |
| 295 | 797c5c00cedd1b91cc97cc37dcbef44 | 29 | 21 | 25 | 29 | 49 |
| 296 | 7b11921e9624d58a2a091c13f358e6f | 21 | 18 | 31 | 35 | 82 |
| 297 | 80ea546b09a879a00496113146b9fe4 | 17 | 17 | 27 | 27 | 52 |
| 298 | 83ce991fc57cb3ca593854c26b6e90d9 | 30 | 25 | 33 | 31 | 280 |
| 299 | 83c57db78a4114319952f44f40be4e80 | 122 | 62 | 141 | 148 | 265 |
| 300 | 8416c4a4f495fe47f5cdce8e8fb674 | 17 | 16 | 28 | 24 | 51 |

| ID | MDS5 | Habo | Padawan | Cuckoo++ | X-Force | PMP |
|-----|------------------------------------|------|---------|----------|---------|-----|
| 201 | 011bb615e582634833c8fb04d04252c | 20 | 16 | 19 | 23 | 519 |
| 202 | 027aabb29c3a3494d78858821555a8b | 31 | 26 | 30 | 36 | 107 |
| 203 | 02f6c231521104b737634d50fa2e9b1819 | 15 | 14 | 42 | 34 | 70 |
| 204 | 03561dd35406b403d85402979b9d05a2 | 43 | 39 | 42 | 43 | 82 |
| 205 | 03625978731ba0f0e28e3dc78343dd968 | 17 | 16 | 28 | 35 | 52 |
| 206 | 03e77d83c324273bee100850e42cd11c | 7 | 15 | 28 | 20 | 139 |
| 207 | 049d713e78333ac6f1b632dce1dd | 15 | 14 | 26 | 27 | 70 |
| 208 | 052666e1f4c9981e7027681563f8867 | 59 | 5 | 58 | 62 | 112 |
| 209 | 0632e98ee1244754c7e914285625ab0 | 216 | 178 | 299 | 216 | 344 |
| 210 | 067329430589b374c35e1b69bad43419 | 21 | 23 | 21 | 27 | 78 |
| 211 | 06a35dd4d6bae273bb42850563e9151fe | 38 | 34 | 37 | 45 | 138 |
| 212 | 07ce36632e2399c1b3218a77599ea771 | 70 | 39 | 99 | 102 | 260 |
| 213 | 07f5bb71414bcb025bbb8014240e80f | 28 | 24 | 27 | 28 | 44 |
| 214 | 08dfbace744477f25f159bc8666a974 | 20 | 16 | 19 | 23 | 563 |
| 215 | 0a44d70788bc1c5f1217f503f2f3ebc8 | 8 | 7 | 20 | 17 | 65 |
| 216 | 0b26005c71c6a142c87f8e976cf704e0 | 72 | 67 | 89 | 72 | 222 |
| 217 | 0b9835f0948498497497833cbl3e212b1 | 26 | 17 | 26 | 26 | 34 |
| 218 | 0d44e50a28c4294570aa834f13d41959 | 15 | 20 | 25 | 15 | 70 |
| 219 | 108079cfc885562a92eb363adbb4182c | 7 | 10 | 17 | 7 | 138 |
| 220 | 11f6f1bb81a837fab5b578332150a78e | 18 | 18 | 28 | 18 | 56 |
| 221 | 125dca5881561fafe56797252d0a39e | 68 | 63 | 70 | 73 | 72 |
| 222 | 135fb83a2a1fad994ac298daa9a27bd | 28 | 23 | 27 | 28 | 42 |
| 223 | 13e0645ba42c32bb049419b83f2dc804 | 17 | 18 | 28 | 30 | 55 |
| 224 | 1408779af2a2e44e736a1f07da29ec8 | 20 | 17 | 19 | 24 | 46 |
| 225 | 14b7884dc5556fe98bd767cd10ac33ca | 49 | 166 | 165 | 183 | 220 |
| 226 | 15b09361380380d3bd4ec7d316b6951 | 306 | 306 | 337 | 340 | 457 |
| 227 | 196360a06b878005a9aae11f5894a34 | 20 | 16 | 19 | 23 | 63 |
| 228 | 1a713da3360a34516ad82b1523abf6d1 | 17 | 17 | 28 | 26 | 64 |
| 229 | 1d21ad688e50c371e8bde9933733d89 | 48 | 46 | 47 | 49 | 84 |
| 230 | 1d541f6ae2474acfd68f79e4aacd1b14 | 20 | 17 | 31 | 27 | 70 |
| 231 | 1db994e8dd0948039693054f683459c | 78 | 89 | 91 | 83 | 104 |
| 232 | 1ed97c5de81a7a9037727c639fat9bfe | 23 | 20 | 22 | 27 | 54 |
| 233 | 1f79632bb62b3497492ec6f6a366d98fc | 349 | 407 | 422 | 419 | 655 |
| 234 | 21c75019e956fca6c34a670c238c379 | 13 | 7 | 22 | 27 | 147 |
| 235 | 2361605b95af66514d0856621854dd26 | 48 | 45 | 47 | 52 | 84 |
| 236 | 2370ef9db483c20f481441a246ab3b2 | 346 | 208 | 354 | 346 | 582 |
| 237 | 256ad8688ea17b514230497d62b8907 | 15 | 15 | 26 | 26 | 14 |
| 238 | 25a5284bc099e246566e0a927fda27fa | 17 | 18 | 28 | 31 | 39 |
| 239 | 292d124aa58579e18239951f63c38da7 | 144 | 129 | 205 | 166 | 300 |
| 240 | 295370e3a3atd08f6babbdf74837f0b | 49 | 45 | 53 | 66 | 84 |
| 241 | 2995574a03023e09199b4c54de34d0 | 13 | 12 | 24 | 26 | 38 |
| 242 | 29f518d6fe7de8dff791d110668b912d | 29 | 44 | 66 | 69 | 43 |
| 243 | 2b79e388966bb783ba81e56b490f3b93 | 52 | 47 | 58 | 65 | 81 |
| 244 | 2e940ae9659ff04a025718e765290 | 21 | 18 | 32 | 33 | 43 |
| 245 | 32370b31ab662e23e9ab4ad4d42819aa | 8 | 8 | 20 | 21 | 64 |
| 246 | 331blcca79f04c3ba0c907bcf07224d1 | 38 | 31 | 38 | 40 | 46 |
| 247 | 33at29b0bdeee7ee22f994f44d423a74 | 20 | 16 | 19 | 24 | 31 |
| 248 | 3498ca6576a3ec21cf28840ff4db5e7 | 17 | 17 | 28 | 30 | 35 |
| 249 | 34e4c33ba5e4451c57906b4476724d6a | 15 | 15 | 26 | 15 | 72 |
| 250 | 3518cd0eebef50798acda3387243f16c | 4 | 13 | 15 | 15 | 111 |

| ID | MD5 | Cuckoo | Habo | Padawan | Cuckoo++ | X-Force | PMP |
|-----|-------------------------------------|--------|------|---------|----------|---------|-----|
| 351 | c39c03e276ac9b6f4a502ae9714187a9 | 21 | 18 | 35 | 34 | 27 | 89 |
| 352 | c42554ae195270102855ecc2c01f0145c9b | 72 | 65 | 67 | 99 | 124 | 125 |
| 353 | c505029f242c0452eed10d77f055921b9 | 51 | 47 | 50 | 51 | 89 | 95 |
| 354 | c61880e699640afbb3a3e0ba7a8498b4 | 17 | 18 | 28 | 26 | 22 | 54 |
| 355 | c83844b19515354483c43374c38629 | 29 | 25 | 26 | 33 | 45 | 83 |
| 356 | c87f1455e2a25d3b68ce4bd4bb072ffb | 17 | 18 | 26 | 29 | 22 | 54 |
| 357 | c8d2fbac6021aa58276a21d1c1d9 | 22 | 29 | 46 | 51 | 42 | 74 |
| 358 | cb0494343321347d28b293c14e2d352f | 8 | 8 | 23 | 22 | 28 | 168 |
| 359 | ccac9675244b88ec3797799e826b89ea | 24 | 20 | 25 | 27 | 34 | 45 |
| 360 | cd31835f1e7f191de488e1ea71912dee | 17 | 17 | 17 | 22 | 22 | 63 |
| 361 | cd9b4354782ac9a26d9277d2d119ec6 | 133 | 133 | 168 | 167 | 138 | 449 |
| 362 | cecd4988602315bc02ac91f8d1bf80c3 | 18 | 18 | 28 | 32 | 40 | 57 |
| 363 | cf22e37378d8c280072c51cd13c612 | 75 | 71 | 74 | 103 | 130 | 131 |
| 364 | d73face1dbd45383e74389a1b63a2790 | 15 | 15 | 25 | 26 | 33 | 72 |
| 365 | d766b045d130c0abcsd65be9254866d2 | 20 | 16 | 19 | 23 | 30 | 524 |
| 366 | db5478bd8c50ee4425c3b7aa7a0342 | 19 | 12 | 20 | 22 | 29 | 263 |
| 367 | dbd1c1e6767ad58940a916a5e50783b | 28 | 24 | 27 | 28 | 42 | 61 |
| 368 | de11905db6d7b885a067283669b0789 | 17 | 17 | 27 | 22 | 39 | 55 |
| 369 | dced135ba29ceee86504064bf45c1fd84 | 8 | 8 | 24 | 22 | 29 | 490 |
| 370 | dd1e0191dbb00996cc30f6a17b968657c | 39 | 35 | 39 | 39 | 53 | 63 |
| 371 | de91ad771b5477a3924ac24a830c7b09 | 17 | 16 | 28 | 17 | 18 | 53 |
| 372 | e41f7965c3ba76029e9803274a928cf4 | 67 | 86 | 108 | 82 | 102 | 198 |
| 373 | e4beb0caef120a317c731c5640e284b | 15 | 14 | 25 | 26 | 33 | 71 |
| 374 | e5c66d51421e69088b7095d68f2e9fa | 14 | 11 | 29 | 29 | 37 | 492 |
| 375 | e7130e2a50499c3acdf4e01306f950 | 17 | 18 | 24 | 25 | 17 | 63 |
| 376 | e8f597edd5441bce904b64417658c4bf | 28 | 23 | 27 | 28 | 43 | 63 |
| 377 | ead453a06315b6f702ad302821337fc2 | 20 | 33 | 49 | 45 | 65 | 70 |
| 378 | eadfb2b01702d212f23e1a24252613e9 | 17 | 18 | 24 | 17 | 22 | 65 |
| 379 | ebd790e97b1403772224429d6b89e4 | 43 | 39 | 42 | 43 | 76 | 83 |
| 380 | ec52663c2e836fab94482c345aab9c5c | 24 | 18 | 24 | 29 | 31 | 46 |
| 381 | ed692adcc957fb5424fe6e0c077c132 | 67 | 62 | 66 | 67 | 117 | 118 |
| 382 | ee14c8b9fc8578f3218cd1da1ba46940 | 20 | 23 | 31 | 32 | 41 | 56 |
| 383 | ee92d85933b024e8d82e03cdfacbaaf6 | 28 | 23 | 27 | 28 | 44 | 56 |
| 384 | f0b820b9602eb7c63821df7cefe4ecd | 38 | 34 | 37 | 38 | 56 | 135 |
| 385 | f335f5857f2d30d00811e1b732f0890a | 15 | 14 | 25 | 15 | 16 | 69 |
| 386 | f3c7855a2bc30b9d02bba8960a112ca | 50 | 44 | 52 | 50 | 66 | 261 |
| 387 | f3ff9415d66bab4f4c55d86e94ea1e85 | 319 | 315 | 317 | 327 | 412 | 416 |
| 388 | f70d182ac7bb3d398ae47d38893dc1e2 | 201 | 188 | 203 | 205 | 258 | 268 |
| 389 | f7e9c33108373192527c3af18a107aff | 23 | 19 | 22 | 29 | 37 | 48 |
| 390 | f8b42194cc19f3157a7c07cae0ff188db | 8 | 7 | 23 | 22 | 28 | 168 |
| 391 | fa5c5264f4668f7a40f7576a27cfe78b | 17 | 17 | 25 | 31 | 39 | 68 |
| 392 | fb9e492cd4a4a0b6e7032919c1f5a8df | 20 | 16 | 19 | 23 | 30 | 550 |
| 393 | fc7184960449af6321c144090b3aa2 | 22 | 19 | 21 | 25 | 32 | 54 |
| 394 | fcfb234b912c84e0524a4393c516c78 | 263 | 35 | 283 | 263 | 285 | 298 |
| 395 | fd594009e2a9f7a7015c3c0878cb886 | 18 | 18 | 28 | 32 | 40 | 56 |
| 396 | fe681844084177d14a0a2e549e9893e | 77 | 88 | 89 | 94 | 104 | 228 |
| 397 | fe742579bfbdb885a81fa16c577def7f | 15 | 14 | 26 | 30 | 33 | 73 |
| 398 | febeat981abcf790fb2177d6c67ced7b | 8 | 8 | 21 | 24 | 30 | 66 |
| 399 | ff0c597903c660cc5577e86caed0bauf | 36 | 21 | 35 | 40 | 52 | 62 |
| 400 | ff3ab2043c7a9c8d84ad785bb9301f83 | 15 | 14 | 25 | 26 | 15 | 69 |

| ID | MD5 | Cuckoo | Habo | Padawan | Cuckoo++ | X-Force | PMP |
|-----|-----------------------------------|--------|------|---------|----------|---------|-----|
| 301 | 85c7e24b1c610e95a00e67d645306475 | 21 | 18 | 32 | 32 | 42 | 74 |
| 302 | 867ac455ce27fbd7872d3cafac729b3f | 77 | 71 | 78 | 77 | 139 | 236 |
| 303 | 87113c55c5398c65c8aa7157b5b64f1a | 8 | 5 | 53 | 32 | 41 | 77 |
| 304 | 88ee691721cf6a62724317aa00293278 | 89 | 32 | 98 | 94 | 110 | 252 |
| 305 | 8932bd03ae481b1b6d6af7c97ea2d41a | 8 | 7 | 24 | 15 | 20 | 168 |
| 306 | 893b1eddfcc390b2668d4da3ac725f6c | 51 | 47 | 50 | 55 | 90 | 97 |
| 307 | 8a4da4bee488749a6abe1d272003d15 | 72 | 67 | 73 | 77 | 125 | 126 |
| 308 | 8a82483ea34fd15601009e8a82c34449 | 43 | 39 | 42 | 61 | 75 | 81 |
| 309 | 8ab6624385a7504c1387683b04c5897a | 153 | 169 | 292 | 221 | 274 | 705 |
| 310 | 8d3ea0751f60fd9f8efcdab7e436851 | 51 | 48 | 50 | 71 | 89 | 95 |
| 311 | 8d6e8286ebca2a3e76e7814be4084fd | 18 | 19 | 29 | 18 | 23 | 55 |
| 312 | 8f184c2e09d6c5c19e1edca08050c347 | 42 | 47 | 62 | 57 | 62 | 83 |
| 313 | 9188f0ff6070e2b28b65aa1c396d89835 | 35 | 35 | 35 | 41 | 53 | 78 |
| 314 | 91f5d45b7a24069e9d20b88870e8c40 | 27 | 23 | 26 | 27 | 40 | 98 |
| 315 | 9330d7d3114fc7fba2ee8d03ad6882ec | 17 | 17 | 27 | 31 | 29 | 54 |
| 316 | 937e25c1e059150dab0ee95a3a715262 | 21 | 25 | 31 | 21 | 27 | 75 |
| 317 | 961c824f208dbd57c2c489955830b195 | 30 | 25 | 29 | 30 | 45 | 56 |
| 318 | 969dd70e0dbad704fc93548224ba8a2 | 17 | 17 | 28 | 33 | 39 | 55 |
| 319 | 98440524e333060a337c5a6ceae06b42 | 18 | 18 | 28 | 32 | 41 | 64 |
| 320 | 98579b288581d02dcb2e658119bce5a2f | 68 | 63 | 69 | 70 | 72 | 74 |
| 321 | 9b0ce3bfff1a0a2b5c3407c0b52dab1a6 | 43 | 40 | 42 | 43 | 75 | 81 |
| 322 | 9b6acc301e9e224fe745906cd818889 | 23 | 23 | 33 | 35 | 44 | 58 |
| 323 | 9b7f5a1228f66cb35c75fb774dc8e | 153 | 203 | 246 | 161 | 199 | 655 |
| 324 | 9bb32e8115e3c643ee55dc41e754da73 | 7 | 10 | 17 | 12 | 25 | 139 |
| 325 | a21e520be784afba01b93b20932ce8c | 18 | 18 | 28 | 27 | 23 | 56 |
| 326 | a27383a46448f25db5dfb6e496ab5d7 | 17 | 17 | 27 | 30 | 39 | 55 |
| 327 | a27ee2b214dfb6b5c1574175f0c96b7f | 144 | 129 | 166 | 144 | 177 | 285 |
| 328 | a4d4b5a8426822a6e26141f0a99781a4 | 14 | 14 | 25 | 25 | 32 | 45 |
| 329 | a8c86a50e5613d2284c7e1a0f18e5b72 | 15 | 15 | 25 | 22 | 33 | 73 |
| 330 | a976deb51d295834b03360919d5544ff | 28 | 23 | 27 | 28 | 42 | 61 |
| 331 | a9b6a5e7044ce975dbd4de90245f3938 | 18 | 19 | 29 | 18 | 23 | 56 |
| 332 | aa5bebb84c2baae824782a85c2bde15a | 58 | 52 | 60 | 58 | 77 | 258 |
| 333 | aa646e4158bec48ec4fc745c36664ffc | 26 | 22 | 25 | 26 | 42 | 131 |
| 334 | ab27f9e2b797edactbe91ae01372ad | 28 | 24 | 27 | 28 | 43 | 63 |
| 335 | ac6d049830db2f68ba01425be886d141 | 87 | 83 | 86 | 87 | 146 | 147 |
| 336 | b03c32330edd83d10f23c941ce11412f | 20 | 17 | 19 | 23 | 30 | 41 |
| 337 | b04ab29e9a7a4f6999c1a8e60aebc5f38 | 17 | 17 | 28 | 26 | 22 | 54 |
| 338 | b0c23492048f6c4595c1847381fd5b2 | 20 | 17 | 19 | 20 | 30 | 544 |
| 339 | b1598c6f6e9552h8c0776163793b529e | 18 | 18 | 28 | 31 | 40 | 56 |
| 340 | b27d6fa1312b314449f7e4ea7e85fc685 | 15 | 14 | 25 | 20 | 33 | 70 |
| 341 | b2c4980b6f6ac9de92c92a702ee5176 | 8 | 8 | 21 | 10 | 27 | 62 |
| 342 | b4bc9b6f1c68981bad1cb40e8cf71e97 | 4 | 6 | 15 | 17 | 22 | 112 |
| 343 | b58f043367e6057c979418d1332c38c8 | 14 | 16 | 284 | 162 | 204 | 420 |
| 344 | b8053ad0847830c698b0bdc350200d8 | 17 | 18 | 27 | 30 | 39 | 57 |
| 345 | b85520d42d64d6d051e75b6112253fce | 14 | 13 | 26 | 25 | 32 | 480 |
| 346 | b931748458cf261c7c14fb0441405 | 24 | 20 | 23 | 27 | 34 | 45 |
| 347 | bace65f81615128345982051ca4605f | 15 | 14 | 26 | 15 | 16 | 70 |
| 348 | bc225bc08b9f0b0d014305a9543d | 55 | 54 | 54 | 72 | 92 | 100 |
| 349 | bf2de044dddf43b4313668ad04ccafcb | 38 | 35 | 37 | 38 | 56 | 139 |
| 350 | b1ef696178596e2b801b39618ec4203 | 14 | 13 | 26 | 14 | 15 | 44 |