# Forecasting Malware Capabilities From Cyber Attack Memory Images

Omar Alrawi*, Moses Ike*, Matthew Pruett, Ranjita Pai Kasturi, Srimanta Barua,
Taleb Hirani, Brennan Hill, Brendan Saltaformaggio
*Georgia Institute of Technology*

## Abstract

The remediation of ongoing cyber attacks relies upon timely malware analysis, which aims to uncover malicious functionalities that have not yet executed. Unfortunately, this requires repeated context switching between different tools and incurs a high cognitive load on the analyst, slowing down the investigation and giving attackers an advantage. We present FORECAST, a post-detection technique to enable incident responders to automatically *predict* capabilities which malware have staged for execution. FORECAST is based on a probabilistic model that allows FORECAST to discover capabilities and also *weigh* each capability according to its relative likelihood of execution (i.e., forecasts). FORECAST leverages the execution context of the ongoing attack (from the malware's memory image) to *guide* a symbolic analysis of the malware's code. We performed extensive evaluations, with 6,727 real-world malware and futuristic attacks aiming to subvert FORECAST, showing the accuracy and robustness in predicting malware capabilities.

## 1 Introduction

Cyber attack response requires countering staged malware capabilities (i.e., malicious functionalities which have not yet executed) to prevent further damages [1], [2]. Unfortunately, predicting malware capabilities post-detection remains manual, tedious, and error-prone. Currently, analysts must repeatedly carry out multiple triage steps. For example, an analyst will often load the binary into a static disassembler and perform memory forensics, to combine static and dynamic artifacts. This painstaking process requires *context switching* between binary analysis and forensic tools. As such, it incurs a high cognitive load on the analyst, slowing down the investigation and giving the attackers an advantage.

To automate incident response, symbolic execution is promising for malware code exploration, but lacks the prior attack execution state which may not be re-achievable after-the-fact (e.g., concrete inputs from C&C activity). Environment-specific conditions, such as expected C&C commands, limit dynamic and concolic techniques (e.g., [3]–[14]) from predicting inaccessible capabilities. In addition, these techniques depend on dissecting a standalone malware binary or running it in a sandbox. However, malware are known to delete their binary or lock themselves to only run on the infected machine (hardware locking). Worse still, researchers found that fileless malware incidents (i.e., only resides in memory) continue to rise [1], [15], [16].

Having access to the *right* execution context is necessary to guide malware into revealing its capabilities. Malware internally gather inputs from environment-specific sources, such as the registry, network, and environment variables, in order to make behavior decisions [11], [17], [18]. Therefore, an ideal and practical input formulation for malware can be adapted from this internal execution state *in memory* bearing the already-gathered input artifacts. It turns out that anti-virus and IDS already collect memory images of a malicious process after detecting it [19]–[21]. A malware memory image contains this internal concrete execution state unique to the specific attack instance under investigation.

During our research, we noticed that if we can *animate* the code and data pages in a memory image, and perform a *forward* code exploration from that captured snapshot, then we can *re-use* these *early* concrete execution data to infer the malware's next steps. Further, by analyzing how these concrete inputs induce paths during code exploration, we can predict which paths *are more likely* to execute capabilities based on the malware's captured execution state. Based

*Authors contributed equally.

on this idea, we propose *seeding* the symbolic exploration of a malware's pre-staged paths with concrete execution state obtained via memory image forensics. Through this, we overcome the previous painstaking and cognitively burdensome process that an analyst must undertake.

We present FORECAST, a post-detection technique to enable incident responders to forecast what capabilities are possible from a captured memory image. FORECAST ranks each discovered capability according to its probability of execution (i.e., forecasts) to enable analysts to prioritize their remediation workflows. To calculate this probability, FORECAST *weighs* each path's relative usage of concrete data. This approach is based on a formal model of the *degree of concreteness* (or $D_C(s)$) of a memory image execution state ($s$). Starting from the last instruction pointer (IP) value in the memory image, FORECAST explores each path by symbolically executing the CPU semantics of each instruction. During this exploration, FORECAST models how the *mixing* of symbolic and concrete data influences path generation and selection. Based on this *mixing*, a "concreteness" score is calculated for each state along a path to derive *forecast* percentages for each discovered capability. $D_C(s)$ also optimizes symbolic analysis by dynamically adapting loop bounds, handling symbolic control flow, and pruning paths to reduce path explosion.

To automatically identify each capability, we developed several modular capability analysis plugins: Code Injection, File Exfiltration, Dropper, Persistence, Key & Screen Spying, Anti-Analysis, and C&C URL Connection. Each plugin defines a given capability in terms of API sequences, their arguments, and how their input and output constraints connect each API. FORECAST plugins are portable and can easily be extended to capture additional capabilities based on the target system's APIs. It is worth noting that FORECAST's analysis only requires a forensic memory image, allowing it to work for fileless malware, making it well-suited for incident response.

We evaluated FORECAST with memory images of 6,727 real-world malware (including packed and unpacked) covering 274 families. FORECAST renders accurate capability forecasts compared to reports produced manually by human experts. Further, we show that FORECAST is robust against futuristic attacks that aim to subvert FORECAST. We show that FORECAST's post-detection forecasts are accurately induced by early concrete inputs. We empirically compared FORECAST to S2E [6], angr [22], and Triton [23] and found that FORECAST outperforms them in identifying capabilities and reducing path explosion. FORECAST is available online at: `https://cyfi.ece.gatech.edu/`.

## 2 Overview

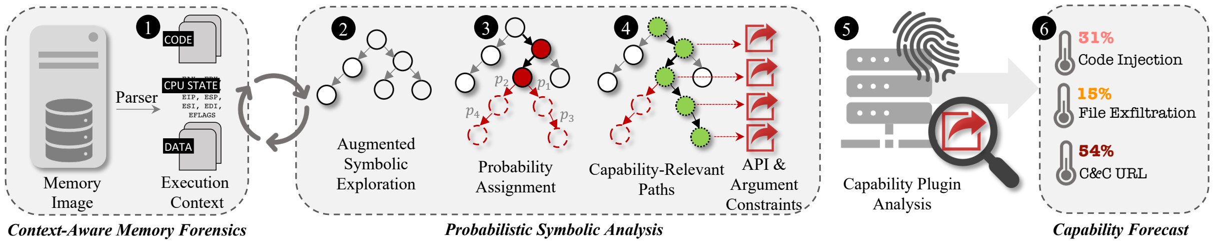
This section presents the challenges and benefits of combining the techniques of memory image forensics and symbolic analysis. Using the *DarkHotel* incident [2] as a running example, we will show how incident responders can leverage FORECAST to expedite their investigation and remediate a cyber attack.

**Running Example - DarkHotel APT.** *DarkHotel* is an APT that targets C-level executives through spear phishing [2]. Upon infection, *DarkHotel* deletes its binary from the victim's file system, communicates with a C&C server, injects a thread into Windows Explorer, and ultimately exfiltrates reconnaissance data. When an IDS detects anomalous activities on an infected host, an end-host agent captures the suspicious process memory (i.e., *DarkHotel's*), terminates its execution, and generates a notification. At this point, incident responders must quickly understand *DarkHotel's* capabilities from the different available forensic sources (network logs, event logs, memory snapshot, etc.) to prevent further damages.

Dynamic techniques [11]–[14] may require an active C&C, which may have been taken down, to induce a malware binary to reveal its capabilities. Because *DarkHotel* only resides in memory, these techniques, which work by running the malware in a sandbox, cannot be applied.[1] With only the memory image, an analyst can use a forensic tool, such as Volatility [24], to "carve out" the memory image code and data pages. Based on the extracted code pages, symbolic analysis can simulate the malware execution in order to explore all potential paths. Unfortunately, existing symbolic tools require a properly formatted binary and are not optimized to work with memory images [7], [22], [23].

Ideally, an analyst can manually project these code fragments into symbolic analysis and source concrete values from the data pages to tell which code branch leads to a capability. However, this back-and-forth process of "stitching up" code with extracted memory artifacts, involves context switching between symbolic execution and the forensic tool. This places a very high cognitive burden on the analyst. An analyst must also handle challenges such as path explosion, API call simulation [4], [22], [25]–[27], and concretizing API arguments (e.g., attacker's URL), which may not be statically accessible in the memory image. Lastly, an analyst must manually inspect APIs along each path to infer high-level capabilities.

[1]Forensic memory images are not re-executable due to being "amputated" from the original operating system and hardware.



1
CODE
CPU STATE
EIP, ESP, ESI, EDI, EFLAGS
DATA
Parser
Memory Image
Execution Context
Context-Aware Memory Forensics
2
Augmented Symbolic Exploration
3
$p_2$
$p_1$
$p_4$
$p_3$
Probability Assignment
4
Capability-Relevant Paths
API & Argument Constraints
Probabilistic Symbolic Analysis
5
Capability Plugin Analysis
6
31%
Code Injection
15%
File Exfiltration
54%
C&C URL
Capability Forecast

Figure 1: FORECAST workflow. A memory image is used to reconstruct the original execution state. Concrete data is utilized to explore code paths while API constraints are analyzed against plugins to forecast capabilities.

## 2.1 Hybrid Incident Response

Incident responders rely on memory forensics to identify attack artifacts in memory images. However, memory forensics alone, which is largely based on signatures, misses important data structures due to high false negatives [21]. On the other hand, symbolic execution can explore code in the *forward* direction, but suffers from issues such as path explosion [22]. To address these limitations, FORECAST combines symbolic execution and memory forensics through a feedback loop to tackle the shortcomings of both techniques.

**Context-Aware Memory Forensics.** Symbolic analysis provides code exploration context to accurately identify data artifacts that are missed by memory forensics. For example, traditional forensic parsing of *DarkHotel*'s memory image missed C&C URL strings because they are obfuscated via a custom encoding scheme. However, subsequent symbolic analysis of the instructions that reference those bytes as arguments, such as a *strncpy* API, allowed FORECAST to correctly identify and utilize these data artifacts in the memory image. Moreover, targeted malware may employ tactics that aim to subvert FORECAST, using anti-forensics and anti-symbolic-analysis, which we carefully considered in our design and evaluation.

Memory image forensics provides concrete inputs that can help symbolic analysis perform address concretization, control flow resolution, and loop bounding. In addition, memory forensics identifies loaded library addresses in memory which allows FORECAST to perform library function simulation.

**Path Probability.** Given a memory image, the goal is to utilize available concrete data to explore potential code paths and forecast capabilities along them. By analyzing how different paths are induced by concrete memory image data, FORECAST can derive the *probability* that a path will reach a capability relative to other paths. FORECAST computes this probability based on modeling how concrete and symbolic data operations are influencing path generation and selection. FORECAST also leverages this probability metric as a heuristic in pruning paths with the least concrete data.

## 2.2 Incident Response with FORECAST

FORECAST identifies capabilities originating from a malware memory image in an automated pipeline. To demonstrate this, we simulated *DarkHotel*'s attack and memory capture, which involved setting up an IDS with DarkHotel's network signature and executing the Advanced Persistent Threat (APT). Following detection, the IDS signals the end host agent to capture the *DarkHotel* process memory. We then input this memory image to FORECAST for analysis. In 459 seconds, FORECAST reveals *DarkHotel*'s capabilities: a C&C communication (i.e., *mse.vmmnat.com*), a file exfiltration (i.e., of host information), and a code injection (i.e., into Windows Explorer).

There are six stages for processing a forensic memory image shown in Figure 1. ① FORECAST forensically parses the memory image and reconstructs the prior execution context by loading the last CPU and memory state into a symbolic environment for analysis. In analyzing the memory image, FORECAST inspects the loaded libraries to identify the exported function names and addresses. Next, ② FORECAST proceeds to explore the possible paths, leveraging available concrete data in the memory image to concretize path constraints. ③ FORECAST models and weighs how each path is induced by concrete data and assigns a probability to each generated path. ④ FORECAST then uses this probability as a weight to adapt loop bounds and prune false paths, allowing FORECAST to narrow-in on the induced capability-relevant paths. ⑤ FORECAST matches identified APIs to a repository of capability analysis plugins to report capabilities to an analyst. Finally, ⑥ FORECAST identifies three capabilities and derives their forecast percentages from the path probabilities as 31%, 15%, and 54%, respectively.

The first path matches the Code Injection plugin. This path contains the APIs: *VirtualAllocEx*, *WriteProcessMemory*, and *CreateRemoteThread*, which are used in process injection. Analyzing the argument constraints leading to these APIs reveals *explorer.exe* as the target process. The second path matches the File Exfiltration plugin. This path contains APIs *getaddrinfo*, *SHGetKnownFolderPath*, *WriteFile*, *Socket*, and *Send*. Forecast inspects their arguments' constraints to determine that the malware writes host information to a file, which it sends over the network. The File Exfiltration plugin concretizes the argument of *SHGetKnownFolderPath* to reveal the file location identifier: *FOLDERID_LocalAppData*. The third path matches the C&C Communication plugin, which reveals a sequence of network APIs including *InternetOpenUrlA*. The plugin queries the API constraints and concretizes *InternetOpenUrlA*'s argument then reports that *DarkHotel* makes an HTTP request to the *mse.vmmnat.com* domain.

Given these forecast reports, an incident responder learns from the captured memory snapshot, that *DarkHotel* will communicate with *mse.vmmnat.com*, steal host data, and inject into Windows Explorer. This will prompt the analyst to block the URL and clean up the affected Explorer process mitigating further damages. Forecast empowers the analyst to quickly and efficiently respond to threats by alleviating the cognitive burden and context switching required to manually obtain the same results.

## 3 System Architecture

Forecast is a *post-detection* cyber incident response technique for forecasting capabilities in malware memory images. It only requires a memory image as input. The output of Forecast is a text report of each discovered capability (e.g., code injection), a forecast percentage, and the target of the capability (e.g., injected process).

**Reconstructing Execution Context.** Forecast parses the memory image to extract the execution state (e.g., code pages, loaded APIs, register values, etc.) to be used to reconstruct the process context. Static analysis of the code pages is used to initialize symbolic exploration. It explores each path beginning from the last IP in the reconstructed process context.

Forecast symbolically executes the CPU semantics of the disassembled code pages until an undecidable control flow is encountered. To resolve this, Forecast recursively follows the code blocks to resolve new CFG paths. When a library call is reached, Forecast simulates and symbolizes the call (discussed in §3.2). Library call simulation introduces symbolic data for each explored state, thus increasing the possibility of state explosion. However, the $D_C(s)$ model (discussed next) provides optimization metrics that enable Forecast to dynamically adapt parameters for loop bounding, symbolic control flow, and path pruning.

### 3.1 Modeling Concreteness to Guide Capability Forecasting

Forecast models how available concrete data in a memory image induces capability-relevant paths using the degree of concreteness model ($D_C(s)$). Degree of concreteness is a property of execution states which encapsulates the "mixing" of symbolic and concrete operations. Symbolic operations ($Sym\_Ops$) make use of symbolic variables such as arithmetic involving symbolic operands. Concrete operations ($Con\_Ops$) do not make use of symbolic variables. $Sym\_Ops$ and $Con\_Ops$ are intrinsic to every state transition. A state transition happens each time a basic block is executed along an explored path. Based on the ratio of $Sym\_Ops$ to $Con\_Ops$, there exists an associated degree of concreteness ($D_C(s)$) value, which measures how *concrete* or *symbolic* the current execution state is.

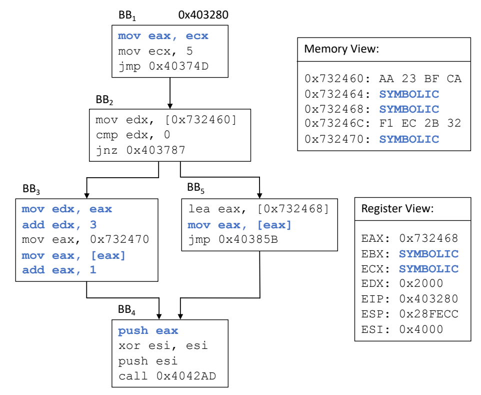
Forecasting is based on malware's use of pre-staged concrete data to execute a set of capabilities. Under $D_C(s)$, paths that increasingly utilize concrete states are more likely to reach a set of capabilities. As a result, Forecast assigns $D_C(s)$ scores to states by modeling their cumulative usage of concrete data. This $D_C(s)$ score is then used to derive the probability, $P_{prob}(s)$, that a path will reach a capability relative to other paths. At the end of exploration, the paths where capabilities are found are analyzed based on their $P_{prob}(s)$, to compute *forecast* percentages of identified capabilities.

In addition to deriving forecasts, $D_C(s)$ detects conditions that trigger path explosion (e.g., rapid path splitting due to symbolic control flows), and makes performance improvements including pruning false states based on the degree of concreteness of every active state (discussed in §3.2).

**Formulation of $D_C(s)$.** For $D_C(s)$ to forecast capabilities, it must summarize two key features: (1) the rate of change in the ratio of symbolic operations to all operations, with respect to state transitions, and (2) the cumulative state conditions from a starting exploration state $j$ to a target state $n$. We normalize $D_C(s)$ with respect to the number of states explored in our model. This bounds its value between 0.0 and 1.0, which describes the current state *mixing*. Formally, we define a state transition set $\tau_n$, which is a set of *ordered* states from $s_j$ to $s_n$:

$$\tau_n := \{s_j, s_{j+1}, s_{j+2}, ..., s_n\} \qquad (1)$$

where state $s_j$ is the first state generated from a memory image and $0 \le j \le n, n \in \mathbb{Z}$. Transitioning from state $s_{i-1}$ to $s_i$ involves executing every operation ($All\_Ops_{i-1}$) in the basic block $BB_{i-1}$ at state $s_{i-1}$. The states in $\tau_n$ are ordered based on the basic block ordering, i.e., the basic block $BB_i$ maps to state $s_i$, and executing $BB_i$



```
BB1 0x403280
mov eax, ecx
mov ecx, 5
jmp 0x40374D

BB2
mov edx, [0x732460]
cmp edx, 0
jnz 0x403787

BB3
mov edx, eax
add edx, 3
mov eax, 0x732470
mov eax, [eax]
add eax, 1

BB5
lea eax, [0x732468]
mov eax, [eax]
jmp 0x40385B

BB4
push eax
xor esi, esi
push esi
call 0x4042AD
```

```
Memory View:
0x732460: AA 23 BF CA
0x732464: SYMBOLIC
0x732468: SYMBOLIC
0x73246C: F1 EC 2B 32
0x732470: SYMBOLIC
```

```
Register View:
EAX: 0x732468
EBX: SYMBOLIC
ECX: SYMBOLIC
EDX: 0x2000
EIP: 0x403280
ESP: 0x28FECC
ESI: 0x4000
```

(a) Symbolic exploration for the control-flow graph, memory, and register values from the memory image.

Let state $s_i$ be the current state after basic block $BB_i$ is executed, and let $D_C(s_i)$ be the degree of concreteness at state $s_i$.

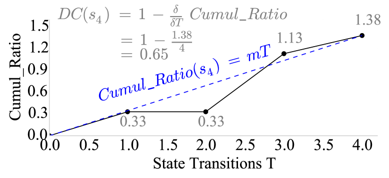
$$D_C(s_1) = 1 - \frac{\frac{1}{3}}{1} \qquad = 1 - \frac{0.333}{1} = 0.67$$

$$D_C(s_2) = 1 - \frac{\frac{1}{3} + \frac{0}{3}}{2} \qquad = 1 - \frac{0.333}{2} = 0.83$$

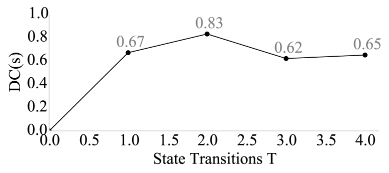
$$D_C(s_3) = 1 - \frac{\frac{1}{3} + \frac{0}{3} + \frac{4}{5}}{3} \qquad = 1 - \frac{1.133}{3} = 0.62$$

$$D_C(s_4) = 1 - \frac{\frac{1}{3} + \frac{0}{3} + \frac{4}{5} + \frac{1}{4}}{4} \qquad = 1 - \frac{1.383}{4} = 0.65$$

(b) Value derivation for degree of concreteness ($D_C(s)$).



(c) Plot of cumulative ratio vs states.



(d) Plot of $D_C(s)$ vs states.

Figure 2: FORECAST recovers context from the process memory image, including the memory values and register values for the captured state in (a). Using the degree of concreteness ($D_C(s)$) formula, (b) calculates the values for each transition state. Figure (c) plots the cumulative ratio of *Sym_Ops* to *All_Ops* accumulated across state transitions. Figure (d) plots the degree of concreteness ($D_C(s)$) across state transitions in the symbolic exploration.

transitions the program's context to $BB_{i+1}$ and state $s_{i+1}$. The set $All\_Ops_i$ is partitioned into 2 disjoint sets, $Sym\_Ops_i$ and $Con\_Ops_i$, such that:

$$Sym\_Ops_i \cup Con\_Ops_i = All\_Ops_i \tag{2}$$

and

$$Sym\_Ops_i \cap Con\_Ops_i = \emptyset \tag{3}$$

For a state $s_n$, we define the $D_C(s_n)$ function as follows:

$$D_C(s_n) = 1 - \frac{\sum_{i=j}^{n} \frac{|Sym\_Ops_i|}{|All\_Ops_i|}}{|\tau_n|} \tag{4}$$

where $|Sym\_Ops_i|$ is the cardinality of the $Sym\_Ops$ performed to reach state $s_i$ and $|All\_Ops_i|$ is the cardinality of $All\_Ops$ performed to reach state $s_i$. Further, $|\tau_n|$ is the cardinality of the state transitions from state $s_j$ to $s_n$.

Tracking the *cumulative ratio* of $Sym\_Ops_i$ to $All\_Ops_i$ for each state transition enables us to calculate $D_C(s)$ instantaneously without iterating through the previous states $s_j$ to $s_n$. An extended form of $D_C(s)$ that allows us to calculate its instantaneous value is given as follows:

$$D_C(s_n) = 1 - \frac{\delta}{\delta T} Cumul\_Ratio(s_n) \tag{5}$$

where, for all transition states $T$, $Cumul\_Ratio(s_n)$ is the sum of the states' ratio for states $s_j$ to $s_n$, and defined as:

$$\{\forall s_i \in T : Cumul\_Ratio(s_n) := \sum_{i=j}^{n} \frac{|Sym\_Ops_i|}{|All\_Ops_i|}\} \tag{6}$$

**An Example of $D_C(s)$ Computation.** Figure 2 is a working example to show the computation of $D_C(s)$. Figure 2a depicts a recovered CFG and memory and register values from the memory image. Symbolic execution starts at basic block $BB_1$ and ends at $BB_4$. We annotate each basic block to show which instructions are *Sym_Ops* based on the register or memory values when the basic block is being executed. Notice that because register *edx* at $BB_2$ and memory address 0x732460 at $BB_2$ have concrete values, only one branch is taken by the conditional jump instructions at the end of $BB_2$. For this reason, $BB_5$ is not explored. Symbolic data can be introduced by I/O-related function calls and calls to functions that are simulated based on FORECAST's function models. Such function calls create symbolic variables within the memory dump which causes a mixing of symbolic and concrete data.

Following along with Figure 2a, Figure 2b computes $D_C(s)$ for each state (basic block) transition. For example, $D_C(s_1) = 0.67$ when we transition to state $s_2$, then it increases to 0.83 as we transition from $s_2$ to $s_3$. For each $D_C(s_i)$ value derived in Figure 2b, we plot them against the transition states in Figure 2d. Figure 2c plots the $Cumul\_Ratio(s_i)$ for each state (shown in black). The instantaneous $Cumul\_Ratio(s_n)$ function is a straight line ($Cumul\_Ratio(s_n) = mT$) drawn from origin to the point $s_n \in T$, where $m$ is the slope. The derivative of $Cumul\_Ratio(s_n) = mT$ gives the instantaneous $D_C(s_n)$ (Equation 5).

**Path Probability.** Given $m$ current states, the path probability of a path $p$, with current state $s$, is derived by dividing $s$'s $D_C(s)$ by the summation of the $D_C(s)$

**Algorithm 1** The Degree of Concreteness ($D_C(s)$)

```
Input: PATHS: Explored program paths in a memory image
Output: $D_C(s)$: $\forall s \in path, \forall path \in PATHS$
                        ▷ Initialize $Cumul\_Ratio$ for each explored path p
for path $p \in PATHS$ do
    $Cumul\_Ratio \leftarrow 0$
    $T \leftarrow 0$
                   ▷ Compute $D_C(s)$ for each state $s$ generated along p
    for State $s \in SuccessorStates(p)$ do
                                          ▷ Get $Sym\_Ops$ and $All\_Ops$
        $Num\_all\_ops \leftarrow GetNumAllOps(s)$
        $Num\_sym\_ops \leftarrow GetNumSymOps(s)$
        ▷ Calculate the ratio of $Sym\_Ops$ to $All\_Ops$ for state s
        $Sym\_Ratio \leftarrow Num\_sym\_ops/Num\_all\_ops$
                        ▷ Update $Cumul\_Ratio$ along the explored path
        $Cumul\_Ratio \leftarrow Cumul\_Ratio + Sym\_Ratio$
                           ▷ Compute $D_C(s)$ for the considered state s
        $D_C(s) \leftarrow 1 - (Cumul\_Ratio/T)$
        $T{+}{+}$
    end for
end for
```

of all $m$ states. This bounds its value between 0.0 and 1.0, and is given as follows:

$$\{P_{prob}(s_x) = \frac{D_C(s_x)}{\sum_{i=1}^{m} D_C(s_i)}, m = |\{AllCurrentStates\}|\} \quad (7)$$

**Algorithmic Approach to $D_C(s)$.** In order to derive $D_C(s)$, FORECAST uses Algorithm 1. *Cumul_Ratio* is the cumulative ratio of symbolic operations to all operations, and $T$ is the total state transitions in terms of basic blocks. For each explored path $p$ in the memory image, $D_C(s)$ is calculated for every state $s$ generated and executed along the path $p$.

## 3.2 DC(s)-Guided Symbolic Analysis

FORECAST uses $D_C(s)$ to optimize symbolic execution multi-path exploration by bounding loops, concretizing addresses for symbolic control flow, and pruning paths. Neglecting these parameters impacts soundness and performance [27], [28]. State-of-the-art tools [6], [22], [23] rely on hard-coded thresholds to balance the *trade-off* between coverage and soundness. These techniques mostly focus on finding bugs in non-malicious code. Choosing an *informed* threshold is application-specific and may require a manual investigation. Yet, unlike finding bugs, malware employ adversarial means to vary these issues at run-time, hence a hard-coded or manual threshold will be limiting. However, by modeling the changing concrete state of an exploration, FORECAST can dynamically adapt these (otherwise application-specific) thresholds at run-time. $D_C(s)$ embodies this automated adaptability to optimize exploration. We evaluate these features against adversarial symbolic analysis tactics in §4.

**Adapting Loop Bounds.** FORECAST optimizes loops by forcing a bound only when $D_C(s)$ indicates a heavy symbolic state over time (specifically, when $D_C(s)$ drops below 0.10 after 10 state transitions). This optimization precisely measures how much a loop is affecting a state to decide when to bound it. We observe that unlike harmless loops, explosion-causing loops converge $D_C(s)$ to 0.10 after two or more transitions.

**On-Demand State Pruning.** When performance is overwhelmed by heavy state symbolism, FORECAST prioritizes states for pruning by selecting the worst performers. Under $D_C(s)$, this selection is trivial since every state has a $D_C(s)$ score, which is used to prune states with heavy symbolic footprints. In §4.6, we found on-demand pruning drove FORECAST toward more concrete paths than tools which prune paths via a hard-coded threshold — leading to FORECAST exploring deeper in selected paths.

**Stack Backtrace Analysis.** False successor paths often arise in symbolic analysis. FORECAST examines the return addresses on the stack in a memory image to identify *false* paths — function returns which do not conform to previously established targets in the call stack. Specifically, the stack backtrace enables FORECAST to verify *flow-correctness* by comparing the stack pointer and return addresses in the backtrace with that computed after executing a return instruction.

**Address Concretization.** FORECAST uses the memory image data space to concretize symbolic indices to a tractable range. In addition, we observed that false states perform illegal indices accesses (indices beyond the mapped code/data space of a process). FORECAST uses this indicator to prune such states. Further, FORECAST's analysis is transparent to address space layout randomization (ASLR) because ASLR is done at process load, before execution.

**Library Function Simulation.** FORECAST analyzes the libraries present in the memory image to identify the exported functions. Identified functions are hooked to redirect the symbolic exploration to a simulated procedure. FORECAST also handles dynamic library loading by calls to the *LoadLibrary* functions. If a library is loaded during symbolic exploration, FORECAST creates a new section in memory for the loaded library. Once a call to *GetProcAddress* is reached, a new address is allocated in the library's memory section and hooked, then this address is returned. Any calls made to this address will be redirected to the correct simulated procedure.

## 3.3 Forecasting Malware Capabilities

To characterize high-level capabilities, we focus on contextualizing a malware's API functionality by analyzing the constraints on their input and output parameters. FORECAST analyzes the symbolic constraints on the input and output parameters of each API to "connect the dots" between APIs. Analyzing APIs used by malware is useful for identifying its capabilities because a malware's behavior stems from its API calls and data flow [11]–[13], [29]. Specifying a unique trace involves identifying the first (*source*) and last (*sink*) API in the sequence. While analyzing API data flow is not novel [30], previous work relies on dynamic taint-tracking [11], [14], [29], which can hardly be applied here. To tackle this, we leverage a *constraint matching* technique introduced by [5] to model malware's decision making. Our approach is based on the formulation that for a given API trace to embody a capability, the path constraints on the input of each *succeeding* API starting from the sink, can be matched to the output constraints of at least one *preceding* API.

When a sink is encountered, FORECAST performs a *call-based* backward slice to record all call instructions such that, for each instruction, there is a data flow from at least one of its operands to the input argument of the *sink*. If the extracted slice includes a corresponding *source*, FORECAST proceeds to match the constraints on the input of every succeeding call, starting from the *sink*, to the output of any preceding call. Note that traditional system call/API tracing often misses malware capabilities due to a lack of contextual connection between observed APIs. Instead, FORECAST uses the constraints on the API parameters in this *call-based* backward slice to precisely connect the data flow between the APIs to infer capabilities. Put simply: The constraints encapsulate only the relevant data flow between sources and sinks.

Figure 3 illustrates this analysis on AveMaria, a Trojan that steals Firefox cookie files. AveMaria infects by replacing the code of *Svcshost*, a Windows service, with its own code, a code injection capability known as *process hollowing*. AveMaria also takes screenshots to spy on the user's screen. The shaded boxes are the relevant APIs in the trace and their key arguments. The dotted line matches the input constraints on an argument of a latter API to the output constraints of at least one preceding API. The analysis starts when a sink is identified (e.g., *SetThreadContext* for AveMaria's Code Injection) and the entire trace is recovered by a call-based backward slice. The numbers, 1, 2, etc., show the constraint matching steps, starting from the *sink* and walking backwards to a *source*. In AveMaria's File Exfiltration, the constraints on the input file ($buf\_3$)

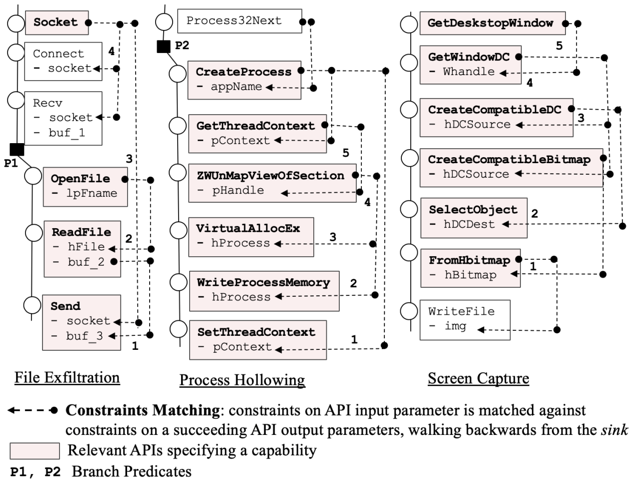


Figure 3: API Constraints Analysis of AveMaria.

exfiltrated by `send` are matched with the constraints on $buf\_2$, an output argument of *ReadFile*. Next, the constraints on the file handle (*hFile*) of *ReadFile* are matched with the constraints on the output of *OpenFile*. When these constraints are matched from a *send* to *socket*, FORECAST reports a File Exfiltration.

**Capability Analysis Plugin.** A plugin specifies different ways that a given capability is to be identified.[2] It lists one or more API sequences, their key arguments, and how constraints on their input and output parameters connect each other. We develop plugins to identify 7 specific malware capabilities. Analysts can easily extend these plugins to specify additional capabilities by reviewing the API documentation of the target operating system. Next, we describe each capability, showing how a plugin can specify them.

**1. File Exfiltration.** Malware sends stolen information from an infected host by uploading a file to its drop site. This is done by using *OpenFile* and *ReadFile* APIs to copy data into a buffer followed by use of the *send* or *HttpSendRequest* network API.[3] The plugin matches the constraints on the buffer written to by *ReadFile* with the buffer of data sent by *send* or *HttpSendRequest*. Figure 3 shows FORECAST's analysis of AveMaria's file exfiltration.

**2. Code Injection.** Malware injects its code into a victim process to run under the target process ID. This is done by the *OpenProcess* or *CreateProcess* APIs, followed by *WriteProcessMemory* (process hollowing) and/or *CreateRemoteThread* (PE or DLL Injection).

[2]Several plugins could be defined for one capability to capture different possible ways that malware exhibit that capability.

[3]We refer to APIs with multiple variants (*A*, *ExA*, *W*, and *ExW*) by the base API name but our plugins cover all variants.

The plugin matches the input constraints on the process handle used by these APIs. Figure 3 shows FORECAST's analysis of AveMaria's code injection.

**3. Dropper.** Malware writes a file to disk and changes its attributes for execution. The plugin matches the constraints on the file handle returned by *CreateFile* with the file handle input passed to *WriteFile*, as well as the file name passed to *CreateFile*, *SetFileAttributes*, and *CreateProcess*.

**4. Key & Screen Spying.** Malware records keystrokes and screenshots of a user's computer. To detect key spying, the plugin matches the constraints on the window handle passed to *RegisterHotKey* and *GetMessage* and checks if *WH_KEYBOARD* was passed to *SetWindowsHook* to monitor keystrokes. For screenshots, the plugin checks if a device context handle returned by *GetDC* or *GetWindowDC* is passed to *CreateCompatibleBitmap*. Figure 3 shows this analysis for AveMaria's screen spying.

**5. Persistence.** Malware make registry entries to maintain persistence across reboots. The persistence plugin compares the constraints on the registry key handle returned by *RegCreateKey* or *RegSetValue* with the input to *RegSetValue*. We also specify the keys and subkeys that malware commonly use with these APIs, such as *HKLM*, *HKCU*, *Run*, and *ControlSet*.

**6. Anti-analysis.** Malware check for analysis environments and tools to determine if it should hide its behavior. This can be done by checking for debuggers with *OutputDebugString*, *IsDebuggerPresent*, or *CheckRemoteDebuggerPresent*. VM checks look for running services by using *CreateToolhelp32Snapshot* or *EnumProcesses* or invoking *cpuid* to check for virtual CPUs. The plugin checks for usage of these APIs.

**7. C&C Communication.** This plugin checks the arguments of *socket* (*af* is an IP address), *InternetOpenUrl* (*lpszUrl* is a domain), and *IWinHttpRequest::Open* (*lpszServerName* is a domain or IP) to determine which servers are contacted. For domains that are represented by constant values or stored in memory (e.g., obtained from an external source such as file or socket), the plugin can successfully extract the domain. If the domain is from an external source and had not be stored in memory at the time of the memory capture, the plugin is unable to determine its concrete value. In the case of domains generated algorithmically, FORECAST builds constraints on the bytes of the domain, seeds Z3 with the concrete execution data, and attempts to solve the constraints.

To develop these plugins, we manually analyzed 50 samples and compiled many relevant API traces and their key arguments, similar to what an analyst would do. Since there are a finite number of ways malware can exhibit a given capability, we can expect to model most of those methods. In doing this, we observed that there could be variations in API traces for the same capability, but the key APIs are always present. In addition, some APIs perform the same function, and hence can be interchanged. For example, $WriteVirtualMemory$ can be interchanged for $WriteProcessMemory$ in the *process hollowing* example in Figure 3. Furthermore, this approach is resilient to noisy API calls that malware authors may mix into their capability function. We provide additional details about the constraints for each plugin in Appendix A Table 7.

**Capability Forecasts Percentages.** The paths where capabilities are found are known as *capability paths* or $C_{Paths}$. FORECAST considers these paths to derive forecast percentages for discovered capabilities. For each capability $c_x$ along a path $x$, FORECAST reports a forecast $C_{cast}(c_x)$ as a percentage. $C_{cast}(c_x)$ is derived from path probabilities of all $C_{Paths}$, and measures the probability that $c_x$ will be executed relative to other capabilities. Let the cardinality of $C_{paths}$ be $m$. A forecast is given as follows:

$$\{\forall i \in C_{Paths} : C_{cast}(c_x) = \frac{P_{prob}(x)}{\sum_{i=1}^{m} P_{prob}(i)} \times 100\} \tag{8}$$

## 4 Evaluation

FORECAST builds upon several angr [22] features, including exploration techniques, SimProcedures, and state plugins. Our focus is on Windows malware since they are most prevalent, but our methodology could be ported to other platforms.

**Experiment Setup.** Our experiment mimics a real-world deployment where a host-based security tool captures a memory image of malware once an IDS detects malicious network activity. Our testbed is comprised of (1) an Ubuntu 14.04 machine (with 40GB RAM and 4-core 2.7GHz cpu) running FORECAST, (2) a Windows 7 machine executing malware, and (3) an IDS system running SNORT. We collected the *alert* network signatures of each malware to configure SNORT. IDS alerts during the malware's execution trigger the capture of a process memory image[4] and sends it to FORECAST. We profiled all captured memory images and observed that 83% were taken while the malware was polling on I/O, such as a network socket.

[4]WinDBG memory capture also collects pages swapped to disk.

| Malware | C&C Comm | | | File Exfiltration | | | Code Injection | | | Dropper | | | Key & Screen Spy | | | Persistence | | | Anti-analysis | | | $O_{FP}$ | $O_{FN}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $P_F$ | $O_M$ | $O_F$ | $P_F$ | $O_M$ | $O_F$ | $P_F$ | $O_M$ | $O_F$ | $P_F$ | $O_M$ | $O_F$ | $P_F$ | $O_M$ | $O_F$ | $P_F$ | $O_M$ | $O_F$ | $P_F$ | $O_M$ | $O_F$ | | |
| Bokbok | 38% | 2 | 2 | 5% | 3 | 3 | 57% | 1 | 1 | | - | | | - | | | - | | | - | | 0 | 0 |
| AcridRain | 23% | 3 | 3 | 19% | 4 | 4 | | - | | 28% | 2 | 2 | | - | | 30% | 1 | 1 | | - | | 0 | 0 |
| AthenaGo | | - | | 11% | 4 | 4 | | - | | 22% | 3 | 2 | | - | | 33% | 2 | 3 | 34% | 1 | 1 | 2 | 0 |
| Rokrat | 30% | 1 | 1 | 26% | 2 | 2 | 22% | 3 | 3 | | - | | 17% | 4 | 4 | | - | | 15% | 5 | 5 | 0 | 0 |
| AdamLocker | 22% | 3 | 3 | 0 | 4 | ∅ | 45% | 1 | 1 | | - | | | - | | 33% | 2 | 2 | | - | | 0 | 1 |
| Marap | | - | | 46% | 3 | 3 | 40% | 1 | 1 | | - | | 14% | 2 | 2 | | - | | | - | | 0 | 0 |
| ATI | | - | | | - | | | - | | 41% | 2 | 2 | | - | | 42% | 1 | 1 | 17% | 3 | 3 | 0 | 0 |
| TeslaAgent | 11% | 4 | 4 | 14% | 3 | 3 | 32% | 1 | 1 | | - | | 13% | 2 | 3 | 30% | 3 | 3 | | - | | 0 | 0 |
| Andromeda | 25% | 2 | 2 | | - | | 14% | 3 | 3 | | - | | | - | | 61% | 1 | 1 | | - | | 0 | 0 |
| AveMaria | 28% | 3 | 4 | 29% | 2 | 2 | 28% | 4 | 3 | | - | | 25% | 1 | 1 | 0 | 3 | ∅ | | - | | 2 | 1 |
| Aveo | 22% | 3 | 3 | | - | | | - | | 40% | 1 | 1 | | - | | 38% | 2 | 2 | 0 | 4 | ∅ | 0 | 1 |
| 7Honest | | - | | 16% | 3 | 3 | 51% | 1 | 1 | 11% | 4 | 4 | | - | | 22% | 2 | 2 | | - | | 0 | 0 |
| Abaddon | | - | | 26% | 2 | 2 | | - | | | - | | | - | | 84% | 1 | 1 | | - | | 0 | 0 |
| AVCrpyt | 51% | 1 | 1 | | - | | | - | | | - | | | - | | 19% | 3 | 3 | 30% | 2 | 2 | 0 | 0 |

Table 1: Capability Forecasts of 14 Select Recent Samples. $P_F$: Forecast percentage, $O_M$: Ground truth manual ordering, $O_F$: FORECAST ordering, $O_{FP}$: Ordering false positive, $O_{FN}$: Ordering false negative.

## 4.1 Evaluating Capability Forecasts

Table 1 presents the capability forecasts of 14 recent samples[5] we manually collected ground truth for. FORECAST output 49 distinct capability forecasts. Manual analysis validated 45 of them; we found 4 false positives (FP) and 3 false negatives (FN), with an accuracy of 86.5%. FPs were due to over-approximating symbolic constraints when simulating undocumented APIs such as *RtlCreateUserThread*. The FNs were due to rare unresolved symbolic targets.

**Ground Truth.** Validating each forecast involves 2 checks: (1) the presence or absence of the identified capability, and (2) the accuracy of the forecast percentage. For ground truth for the presence or absence of a capability, we leveraged malware reports from security vendors [31], [32] and our own manual analysis. We also used the MITRE ATT&CK Framework [33] for our initial ground truth mappings.

To validate our ground truth forecast percentages, (i.e.rank each outcome according to the "difficulty" or "constraints required" of arriving at an outcome) we modeled the difficulty metric of executing capabilities from the memory image capture point based on the number of branch constraints to reach a given capability. We can obtain this metric via manual analysis of the memory image since we know the addresses of the individual capabilities. Using *Bokbot* as an example, Table 1 shows its 3 capabilities: Code Injection, C&C Communication, and File Exfiltration. For these, FORECAST reports forecast percentages of 57%, 38%, and 5% respectively (listed in the $C_{cast}$ column of Table 1). Based on manual analysis of its memory image, the number of branch constraints to reach these capabilities are 166, 195, and 257, respectively. Thus, Code Injection is *less difficult* to reach and hence has the highest forecast.

[5]Their hashes are presented in Table 8 in Appendix A.

Next, we validate capability ordering. We assign an increasing number, starting at 1, to each capability identified by manual checking (defined as $O_M$) and ordered by increasing difficulty. We assign an increasing number to each capability identified by FORECAST ($O_F$) up to the number of identified capabilities. For *Bokbot*, both manual checking and FORECAST report an ordering of 1, 2, and 3 for Code Injection, C&C Communication, and File Exfiltration respectively.

As shown in Table 1, because FORECAST's forecast for *Bokbok*'s Code Injection is the highest, (i.e., 57%), Code Injection's ordering or $O_F$ is 1. Similarly, the ordering by manual checking or $O_M$ is also 1, which validates FORECAST's forecast for *Bokbot*'s Code Injection. In another example, FORECAST's prediction for *AthenaGo*'s Dropper is 22%, which is the second highest forecast (i.e., $O_F$ is 2). However, manual checking shows Persistence as the second highest instead, resulting in FP for *AthenaGo* (listed in the $O_{FP}$ column of Table 1). FORECAST missed *Aveo*'s Anti-analysis capability, resulting in a FN (listed in the $O_{FN}$ column), and a forecast of 0 ($C_{cast}$ column).

Overall, Persistence reported the highest forecast percentages, as high as 84% for Abaddon. We found that most malware persist via infecting the registry. Conversely, File Exfiltration reports the lowest forecasts, as low as 5% for Bokbok. Reasonably, File Exfiltration can be seen as an "end goal" capability, which malware deploy in deep code under several constraints. By integrating capability analysis plugins, FORECAST was able to automatically identify them.

**C&C Communication.** Table 1 shows 7 C&C domains identified with 1 FP. We focused on WinINET's APIs such as *InternetOpenUrl* and *socket*. In particular, we concretized their domain and IP address arguments. FORECAST revealed *Rokrat* and

| | | | | | | | | | Capabilities | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Malware | | Packer | Paths | Steps | Const. | Leaves | Time (s) | $D_C(s)$ | C&C | Exfil. | Inject | Drop | Spy | Persist | Anti-Analy. | FP | FN |
| **Packed From Table 1** | Marap | UPX Type-I | 227 | 465.95 | 25.74 | 3.01 | 97.39 | 0.94 | | ✓ | ✓ | | ✓ | | | 0 | 0 |
| | AVCrypt | | 59 | 184.69 | 23.53 | 2.00 | 27.91 | 0.84 | ✓ | | | | | ✓ | ✓ | 0 | 0 |
| | ATI | | 115 | 179.44 | 19.89 | 3.17 | 56.90 | 0.83 | | | | ✓ | | ✓ | ✓ | 0 | 0 |
| **Stress Test Packers** | RokRat | ASPack Type-III | 595 | 265.68 | 14.05 | 1.99 | 143.54 | 0.93 | ✓ | ✗ | ✓ | | ✓ | | ✓ | 0 | 1 |
| | AcridRain | | 1410 | 330.39 | 26.82 | 2.84 | 247.47 | 0.88 | ✓ | ✓ | | ✓ | | ✗ | | 0 | 1 |
| | AthenaGo | | 677 | 371.39 | 26.48 | 2.03 | 193.44 | 0.92 | | ✓ | | ✓ | | ✓ | ✓ | 0 | 0 |
| | RokRat | Armadillo Type-VI | 732 | 56.39 | 18.19 | 2.96 | 139.31 | 0.68 | ✓ | ✓ | ✗ | | ✓ | | ✓ | 0 | 1 |
| | AcridRain | | 338 | 226.30 | 23.70 | 3.42 | 93.34 | 0.84 | ✓ | ✓ | | ✓ | | ✗ | | 0 | 1 |
| | AthenaGo | | 701 | 55.21 | 18.17 | 2.66 | 107.42 | 0.67 | | ✓ | | ✗ | | ✓ | ✓ | 0 | 1 |

Table 2: Packed malware evaluation results based on packer taxonomy found in Ugarte-Pedrero et al. [34].

*AVCrypt's* usage of `dropbox.com` and TOR (`bxp44w3qwwrmuupc.onion`), respectively. *TeslaAgent* uses a hardcoded IP address (45.77.35.239), and a gmail account (`mylogbox3h@gmail.com`) to communicate externally. Aveo communicates with a `.it` domain, `vacanzaimmobiliare.it`. We found that this server is hosting a vacation website and is likely compromised.

**Code Injection.** FORECAST reports 8 Code Injection with 1 FP. *Explorer* and *Svchost* are the most common Windows programs injected into. 7Honest, Bokbot, and AveMaria hollows into *Svchost* by invoking *CreateProcess* with a `CREATE_SUSPENDED` flag, and thereafter swaps the code pages with *WriteProcessMemory* and *SetThreadContext*. TeslaAgent and Andromeda inject into *Explorer* using the *VirtualAlloc* and *WriteProcessMemory* API sequences.

**Dropper.** FORECAST reports 5 Dropper forecasts with no FP and FN. *7Honest* and *AthenaGo* drop additional files in the `AppData` and `ProgramData` directories and manipulate their permissions using *SetFileAtrributes*. *AcridRain* drops a *WinDDecode.exe* executable in `AppData`. We determined it was a custom decoder for its C&C. *Aveo* drops `.dat` executables in `system32`.

**Key & Screen Spying.** We focused on detecting keyloggers and screen captures based on the *Key Hooks* and *GDI* API toolkit. FORECAST reported 4 Key & Screen spying forecasts with 1 FP. RokRat and TeslaAgent used *GetAsyncKeyState* and *RegisterHotKey* API to obtain key presses. AveMaria invoked screen capture using a sequence of *GetDeskstopWindow*, *GetWindowDC*, and *CreateCompatibleBitmap*.

**Anti-Analysis.** FORECAST reports 4 Anti-analysis forecasts with 1 FN. *RokRat* and *AthenaGo* performed network checks via *InternetCheckConnectionA*. *AVcrypt* uses *IsDebuggerPresent*, *OutputDebugString*, and *CheckRemoteDebuggerPresent* to check for debuggers. To check for VM, ATI issues *cpuid* calls to obtain hardware platform information.

## 4.2 Packed Malware

We evaluated FORECAST's robustness against packers using the taxonomy proposed by Ugarte-Pedrero et al. [34]. In fact, 3 of the 14 samples from Table 1 are packed by UPX, which is a Type-I packer. We include those three samples in our packer robustness evaluation as a reference, as shown in Table 2. Our evaluation also looks at three additional families using two different types of packers, namely *ASPack* (Type-III) and *Armadillo* (Type-VI), giving us a total of 9 samples.

Type-I through Type-IV packers fully unpack the malware code in memory *before* executing the malicious code [34]. For completeness, we evaluate FORECAST against ASPack, a Type-III packer, where layered unpacking routines are not sequential, leaving junk code and data in memory from earlier layers. In Table 2, FORECAST explores an average of 894 paths per sample with a high final $D_C(s)$ (mostly concrete). Additionally, FORECAST identifies almost every capability found in Table 1, except for exfiltration (Exfil.) and persistence (Presist) capabilities for *RokRat* and *AcridRain*, respectively. We mark those missed capabilities as false-negatives (FN) in Table 2.

Type-V and VI packers unpack malicious code incrementally using different memory frames. We evaluate FORECAST against Armadillo with CopyMem-II protection, which incrementally unpacks and executes code at a memory-page granularity. FORECAST explores an average of 590 paths per sample with an average of 0.73 for the final $D_C(s)$, which is lower than Type-I and Type-III packers. Moreover, FORECAST identifies all the capabilities in Table 1 except code injection (Inject), persistence (Persist), and dropper (Drop) capabilities found in RokRat, AcridRain, and AthenaGo, respectively. These results empirical show the effect of incremental unpacking on FORECAST's capability to analyze malware, which is rooted in the memory artifacts that are visible during malware capture. We discuss these limitations in §6.

| **Malware Family** | All Samples | browsefox | coinminer | xtrat | autoit | expiro | bifrose | darkkomet | rebhip | dprotect | llac | delf |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Total Samples | 6,727 | 200 | 161 | 57 | 161 | 3428 | 69 | 163 | 80 | 398 | 68 | 65 |
| C&C URL | 30.5% | 51% | | 47% | | 39% | 32% | | | | 17% | |
| File Exfiltration | 11.3% | 12% | | | | 17% | 8% | | | | | |
| Code Injection | 32.7% | | 25% | | | | | | | | | 44% |
| Dropper | 41.0% | 37% | 23% | 23% | 11% | 44% | | 33% | | 37% | 26% | 10% |
| Persistence | 55.2% | | 52% | | 60% | | 67% | | 61% | 63% | 57% | 46% |
| Key&Screen Spy | 24.4% | | | 40% | | | | 33% | | | | |
| Anti-analysis | 29.4% | | | | 29% | | | 34% | 39% | | | |
| Avg. Explore time(s) | 291 | 218 | 234 | 196 | 124 | 310 | 128 | 326 | 285 | 227 | 134 | 420 |
| Avg. APIs per path | 26 | 21 | 18 | 12 | 9 | 17 | 29 | 13 | 45 | 28 | 13 | 11 |
| Avg. States generated | 1638 | 1196 | 1267 | 3450 | 950 | 1471 | 4601 | 670 | 897 | 1136 | 823 | 1568 |
| $D_C(s)$ of final states | 0.21 | 0.34 | 0.21 | 0.29 | 0.39 | 0.28 | 0.18 | 0.43 | 0.43 | 0.32 | 0.41 | 0.31 |

Table 4: Average Capability Forecasts and Metrics, featuring the 11 most prevalent malware families.

## 4.3 Tactics To Subvert Forecast

| Category | Samples | Paths | Steps | C/P | Leaves | Flags |
|---|---|---|---|---|---|---|
| No Hash | 10 | 2.00 | 21.50 | 3.00 | 16.00 | 100% |
| Hash-Guarded | 10 | 74.70 | 45.15 | 19.00 | 3.40 | 100% |
| Tigress | 2311 | 4.02 | 58.65 | 8.47 | 3.84 | 97% |

Table 3: Averaged results of symbolic obfuscation evaluation. C/P denotes constraints per path.

Malware authors who are aware of Forecast may try to adapt advanced tactics to subvert our capability exploration. To evaluate Forecast against targeted attacks, we follow the set of obfuscation benchmarks proposed by Banescu et al [35], [36]. These anti-analysis benchmarks are broken into two sets, a set of 10 hash-guarded programs that simulate license checking (*Hash-Guarded*) and a set of 2,311 Tigress-obfuscated programs (*Tigress*). Table 3 presents the results for three experiments, namely baseline (*No Hash*), *Hash-Guarded*, and *Tigress*. For the *Hash-Guarded* programs we created a Forecast plugin that triggers when the license check is correct (captured *Flag*). For the *No Hash* programs, Forecast found the flag and explored 2 paths with an average of 21.5 steps per path, 3 constraints, and 16 leaf nodes per constraint AST.

For the *Hash-Guarded* programs, Forecast found all flags and explored an average of 74.7 paths, with 45.15 steps and 19 constraints per path, and 3.4 leaves per constraint. For the *Tigress* obfuscated programs, the code performs various transformations on the input and compares the derived value against an expected value that represents the correct license key.[6] The results show that 97% of the flags were found and an average of 4.02 paths were explored with an average of 58.65 steps per path, 8.47 constraints per path, and 3.84 leaves per constraint. These results empirically demonstrate that Forecast is resilient against adversarial obfuscation attacks targeting symbolic execution.

[6]We excluded Tigress programs which crashed or did not print the flag during a natural execution with the correct argument.

## 4.4 Large-Scale Analysis

We show that Forecast is effective when applied to a larger set of memory images from 6,727 malware samples (covering 274 different malware families). Table 4 summarizes Forecast's capability forecasts for the 6,727 samples and highlights metrics for the top 11 most prevalent malware families in our dataset. The highest capability forecasts were recorded for Persistence and Dropper. We observed that over 70% of all 6,727 samples have Dropper and Persistence capabilities. When averaged, Persistence reports 55.2% overall forecast, peaking at 67% for the *Bifrose* family. Our experiment revealed that the *Bifrose* family enters several registry `Run` keys in both the `HKLM*` and `HKCU*` registry directories – an aggressive means to force persistence across reboots, compared to other families. *Bifrose* samples also drop a `.dat` executable file in `Windows\System32` and connect to a `no-ip.com` domain C&C. Dropper capability reported 41.0% overall, peaking at 44% for the *expiro* family. The lowest forecasts were File Exfiltration, with 11.3% overall.

We observe fairly low variance between the highs and lows of forecasts within each family. Digging deeper, this is due to samples in the same family reusing the same features (e.g., dropper filenames). Samples in the *Browsefox* adware family drop an executable with a consistent file name format of "`<random>Expance.exe`" in `ProgramData` directory. Our investigation found that it installs extensions to browsers to display ads, earning the attacker ad revenue. The *Xtrat* family of remote access trojans displays similar patterns of C&C domain names, namely `<random>to.org`. Concrete examples are `zapto.org` and `hopto.org`.

**Exploration Metrics.** Table 4 reveals interesting observations about the metrics reported by each malware family. The average exploration time for one memory image is 291 seconds, which shows that Forecast is efficient as an offline investigation tool. Forecast revealed an average of 26 unique APIs per memory image and generated 1,638 states on average per sample. The *Bifrose* family reported the largest number of states per sample (4,601), while *Darkkomet* generated the lowest (670 states per sample).

The average $D_C(s)$ for end states was 0.21, which indicates that states toward the end were more *symbolic* than *concrete*. *Bifrose* reported the lowest $D_C(s)$ of 0.18 indicating a very *symbolic* ending. This was the general observation for most C&C-based malware since simulating socket calls introduces more symbolic data, causing $D_C(s)$ to drop. *Darkkomet* and *Rebhip* tied for the highest $D_C(s)$ with 0.43. This confirms the correlation between $D_C(s)$ and cumulative states coverage. Samples in the *Delf* family reported the maximum exploration time (420 seconds on average), which explains their high average states (1568).

## 4.5 Pre-Staged Concrete Input

Recall that when no concrete input data exists, pure symbolic analysis will explore all paths. The $D_C(s)$ model assumes that following paths that involve pre-staged concrete data in the memory image focuses FORECAST on the most urgent payloads. We empirically evaluated this assumption with controlled-experiments on 2 malicious and 3 benign programs: (1) *LokiRAT*, a remote access trojan, (2) *xTBot*, an IRC-based malware, (3) `netstat`, (4) `ipconfig`, and (5) `arp`. These were chosen because their source code is publicly available and their behavior for concrete inputs can be determined.[7] We analyzed their source code and compiled binaries to establish the ground truth set of paths that selected concrete inputs will cause the program to take. For *LokiRAT* and *xTBot*, we determined all specific paths that the malware could take when it receives certain commands from its C&C server. For `netstat`, `ipconfig`, and `arp`, we determined all specific paths that the programs could take when executed with a set of command-line flags. Figure 4 illustrates an example.

Table 5 shows these programs and each of the concrete data we investigated. For `netstat`, `ipconfig`, and `arp`, we executed each program with the command-line flags shown in Table 5 and took a memory image when *main* was called to ensure the flags exist in the memory image as concrete data. For these experiments, we obtained 9 memory images (3 command-line flags for each of the 3 programs). For *LokiRAT* and *xTBot*, we executed each sample, injected each selected C&C command, and captured memory images as soon as they received each C&C command (6 in total). The intuition here is that FORECAST should produce the same paths as each ground truth set for the corresponding memory images.

[7] FORECAST did not have access to the ground truth.

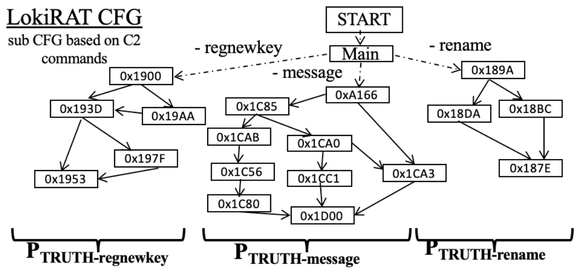


Figure 4: *LokiRAT* ground truth. $P_{TRUTH-regnewkey}$, $P_{TRUTH-message}$, and $P_{TRUTH-rename}$ represent the ground truth set of paths for each *LokiRAT* C&C command (regnewkey, message, rename).

| Malware | Ground Truth | | FORECAST Results | | | | |
|---|---|---|---|---|---|---|---|
| | C&C Cmds | Paths | Paths | TP | FP | FN | Acc(%) |
| LokiRAT | regnewkey | 4 | 5 | 4 | 1 | 0 | 80 |
| | message | 4 | 4 | 4 | 0 | 0 | 100 |
| | rename | 2 | 2 | 2 | 0 | 0 | 100 |
| XTBot | .ntstats | 1 | 1 | 1 | 0 | 0 | 100 |
| | .netinfo | 2 | 2 | 2 | 0 | 0 | 100 |
| | .sysinfo | 28 | 30 | 27 | 2 | 1 | 90.0 |
| **Benign** | **Argument** | **Paths** | **Paths** | **TP** | **FP** | **FN** | **Acc(%)** |
| netstat | -a | 3 | 3 | 3 | 0 | 0 | 100 |
| | -e | 3 | 3 | 3 | 0 | 0 | 100 |
| | -r | 2 | 2 | 2 | 0 | 0 | 100 |
| ipconfig | -release | 4 | 4 | 4 | 0 | 0 | 100 |
| | -renew | 6 | 5 | 5 | 0 | 1 | 83.3 |
| | -no-flag | 19 | 18 | 16 | 2 | 1 | 84.2 |
| arp | -a | 6 | 6 | 6 | 0 | 0 | 100 |
| | -d 10.1.1.1 | 8 | 7 | 7 | 0 | 1 | 87.5 |
| | -s :cf:b8:20 | 11 | 12 | 10 | 1 | 1 | 83.3 |

Table 5: Exploration Based on Pre-Staged Input.

Table 5 shows that, for the malware, FORECAST discovered 40 out of 41 ground truth paths, with 3 FP and 1 FN, giving an accuracy of 95.0%. For the benign programs, FORECAST discovered 56 out of 62 ground truth paths, with 3 FP and 4 FN, giving an accuracy of 93.1%. We found that the FP results were caused by short paths that were pruned when they accessed illegal memory. FNs were caused by symbolic IP values due to unconstrained jump targets. Overall, FORECAST attained an accuracy of 94.0%. This shows that FORECAST's exploration of memory images using pre-staged inputs is accurate.

## 4.6 Comparing Existing Techniques

We empirically compared FORECAST with S2E [6], angr [22], and Triton [23]. We found that FORECAST outperforms them at identifying malware capabilities based on the coverage of capability paths (i.e. code paths where at least one capability is found). Since they cannot take a memory image as input, with the exception of angr, we provided the malware binary and configured them to start from an equivalent IP as

| Tools | Exploration techniques | Explored paths | Identified capabilities | Path explosion instances | Explore time(s) | Basic blocks covered |
|---|---|---|---|---|---|---|
| FORECAST | Data-Guided | 877 | 32 | 28 | 301 | 12488 |
| angr [22] | Pure Symbolic | 1292 | 11 | 521 | 236 | 14567 |
| S2E [6] | Concolic | 602 | 7 | 57 | 98 | 10007 |
| Triton [23] | Concolic | 229 | 3 | N/A | 522 | 4309 |

Table 6: FORECAST Compared to Existing Techniques.

FORECAST. We used 50 samples for this experiment.[8]

As shown in Table 6, FORECAST identified more than twice the capabilities compared to angr, S2E, and Triton. FORECAST explored as many as 877 paths per sample on average. By leveraging prior execution state to optimize paths, only 28 paths were terminated due to path explosion compared to 521 by angr and 57 by S2E. Although angr explored the most paths (1292), it terminated 521 due to path explosion. We observed that angr could not concretize paths when faced with early symbolic control flow, causing state explosion. The exploration time for angr was relatively low (236s) because many paths quickly became unconstrained and terminated. FORECAST reported a higher runtime of 301s due to the overhead of computing probability scores for each state.

S2E requires symbolic variables to be manually induced for multi-path exploration. When we initially tested S2E with malware, we traced only a single path. However, to enable S2E to explore multiple paths, we symbolized the arguments of the malware's local functions and only traced paths that originated from the malware code. This led to an exploration of 602 paths, where 57 became unconstrained and terminated. S2E had the fastest average runtime (98s), because it executes code natively on the CPU. Triton uses a per-input iterative approach to code exploration, hence the path explosion metric is not applicable. To trace multiple paths with Triton, we manually pushed new constraints to each path predicate, but Triton was heavily hindered by input requirements to explore new paths. Triton traced 229 paths on average, 3 of which identified capabilities. Due to its iterative nature and instruction-level emulation, it incurred the highest runtime of 522 seconds.

## 5 Related Work

Prior work uses symbolic execution for various applications including test case generation [8]–[10], [27], [37]–[40], vulnerability detection [9], [41], [42], and enhancing dynamic malware analysis [3], [5], [43], but often relies on simplistic heuristics to optimize symbolic execution. FuzzBall [44] initializes the program states to concretize constraints, while MAYHEM [27] applies on-line and off-line concolic execution to manage path exploration. However, FORECAST reduces path explosion by using the $D_C(s)$ framework to identify capability-relevant paths. Additionally, FORECAST does not require an intact binary file or prior knowledge of a program's input and environment, which avoids restrictive assumptions for symbolic execution.

For malware applications, prior works use full-system emulation [4], dynamic analysis [5], [45], and Win API simulation [46] to identify malware capabilities. Yadegari et al. [47] study the robustness of symbolic analysis techniques against malware obfuscation. In contrast, FORECAST is a post-detection approach that combines both symbolic analysis and memory forensics to identify staged malware capabilities. Prior work on memory forensics focuses on kernel objects [48], [49], access patterns to kernel objects [50]–[54], and dynamic memory traces [55], [56] to detect and remediate rootkit malware. DSCRETE [57] leverages memory image code reuse for interpreting single data structures. Similarly, for mobile security, prior works [58]–[61] analyze a mobile application's memory to recover artifacts related to recent activities. However, FORECAST relies on memory artifacts to contextualize malware behavior through symbolic analysis and surgically analyzes a single target malicious process.

Provenance-based investigation techniques are also related to FORECAST. NoDoze [62] and Hassan et al. [63] utilize Windows and Linux system events to prioritize alerts through a network diffusion approach using temporal ordering. Similarly, HOLMES [64] correlates suspicious events by examining information flows and TARDIS [65] identifies compromised websites through a spatial-temporal approach to present attack tactics for analysts. Attack2Vec [66] uses system event embedding to derive emerging attack tactics. FORECAST uses the $D_C(s)$ model to predict in-progress malware capabilities using a similar network diffusion approach [62]–[64] but instead identifies relevant paths based on the execution context of a malware.

## 6 Limitations and Discussion

**Subverting Symbolic Analysis.** An adversary may target the symbolic execution component of FORECAST by exploiting path explosion, path divergence, and constructing complex constraints. In §4.3, we turned to the published literature on symbolic analysis benchmarks [35] and found that FORECAST is robust against these attacks. However, we acknowledge that a novel attack, not considered in the literature, may subvert FORECAST's results.

[8]Hashes and capability addresses are in Table 9 in Appendix A.

**Subverting Memory Artifacts.** An adversary may target memory acquisition or memory artifacts to subvert FORECAST. The memory acquisition depends on the IDS, which FORECAST has no control over. It is reasonable to assume that the IDS will detect and capture a malware while the malware is executing malicious routines (which produced the detected signature). To tamper with memory artifacts, an adversary can obfuscate code segments, use a non-standard stack layout, or insert junk code/data. FORECAST was shown to be resilient to junk code/data produced by Type-III packers in §4.2. If FORECAST is affected by an attack that subverts code analysis, FORECAST could be extended to handle specific memory manipulation attacks by porting IDA *microcodes* to flatten obfuscated code structures [67].

**Virtualization-Based (VM) Packing.** Generally, like any symbolic exploration framework, FORECAST cannot explore capabilities in packed code unless it is unpacked. As our evaluation in §4.2 shows, FORECAST can handle Type-I, Type-III, and Type-VI packers as outlined in Ugarte-Pedrero et al. [34]. Some packers use virtualization to convert programs into bytecode and use an interpreter to run the bytecode. Due to the complexity of virtualization, FORECAST cannot handle such techniques, which account for less than 2% of packed malware [34].

**Adversarial Aware Attacks.** An adversary that is aware of FORECAST can influence the analysis via two factors: memory frame replacement and pointer obfuscation. First, memory frame replacement can subvert FORECAST for specific samples using *Armadillo* with CopyMem-II protection due to the iterative unpacking and execution sequence. This unpacks code at a memory frame-level granularity, which limits the visibility of the malicious code to the most recent unpacked memory frame. This artifact is evident from our evaluation in Table 2. Second, pointer obfuscation creates additional overhead for the symbolic execution engine, which drops the degree of concreteness ($D_C(s)$) metric. An attacker can heavily utilize pointer obfuscation by relying on a unique seed in memory to deobfuscate pointers.

Heavy obfuscation of memory artifacts can and does affect the performance and stability of the malware, which may not be in the favor of the malware operator. Not surprisingly, Ugarte-Pedrero et al. [34] finds such heavy obfuscation in only 1.8% of in-the-wild malware. Finally, we emphasize that the quality of the memory capture is dependent on the detection tool, independent of FORECAST. FORECAST is a post-detection approach that relies on a forensic memory capture to perform capability prediction.

## 7 Conclusion

FORECAST overcomes the high cognitive burden on an analyst by forecasting future malware capabilities. FORECAST integrates memory forensics and symbolic analysis in a feedback loop to efficiently explore malware with context. Our evaluation has shown that FORECAST produces accurate forecasts of capabilities.

## Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments and feedback. We also thank Professor Nolen Scaife for his guidance while shepherding this paper. This work was supported, in part, by ONR under Award N00014-19-1-2179 and NSF under Award 1755721. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of our sponsors or collaborators.

## References

[1] *Fileless attacks against enterprise networks*, `https://securelist.com/fileless-attacks-against-enterprise-networks/77403/`, 2017.

[2] *The Darkhotel APT: A Story of Unusual Hospitality*, `https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/08070903/darkhotel_kl_07.11.pdf`, 2014.

[3] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, "Dynamic analysis of malicious code," *Journal in Computer Virology*, vol. 2, no. 1, 2006.

[4] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin, "Bitscope: Automatically dissecting malicious binaries," *Technical Report, School of Computer Science, Carnegie Mellon University*, vol. CS-07-133, 2007.

[5] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," in *Proceedings of the 28th Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2007.

[6] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, Mar. 2011.

[7] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[8] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proceedings of the ACM SIGSOFT Software Engineering Notes*, Lisbon, Portugal, Sep. 2005.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," *ACM Transactions on Information and System Security*, vol. 12, no. 2, 2008.

[10] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun. 2005.

[11] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero, "Identifying dormant functionality in malware programs," in *Proceedings of the 31th Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2010.

[12] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell, "A layered architecture for detecting malicious behaviors," in *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2008, pp. 78–97.

[13] K. A. Roundy and B. P. Miller, "Hybrid analysis and control of malware," in *Proceedings of the 13th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Ottawa, Canada, Sep. 2010.

[14] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda, "Inspector gadget: Automated extraction of proprietary gadgets from malware binaries," in *Proceedings of the 31th Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2010.

[15] *Non-Malware Attacks and Ransomware Take Center Stage in 2016*, https://www.carbonblack.com/wp-content/uploads/2016/12/16_1214_Carbon_Black-_Threat_Report_Non-Malware_Attacks_and_Ransomware_FINAL.pdf, 2016.

[16] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Chen, W. Cheng, C. Gunter, and H. Chen, "You are what you do: Hunting stealthy malware via data provenance analysis," in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[17] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2008.

[18] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna, "Efficient detection of split personalities in malware," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2010.

[19] *FireEye: Endpoint Forensics*, https://www.fireeye.com/products/mir-endpoint-forensics.html, [Accessed: 2018-02-28].

[20] B. D. Carrier and J. Grand, "A hardware-based memory acquisition procedure for digital investigations," *Digital Investigation*, vol. 1, 2004.

[21] S. Vömel and F. C. Freiling, "A survey of main memory acquisition and analysis techniques for the windows operating system," *Digital Investigation*, 2011.

[22] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 37th Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[23] *Triton: A Dynamic Symbolic Execution Framework*, SSTIC, 2015, pp. 31–54.

[24] *Volatility: Open Source Memory Forensics Framework*, https://www.volatilityfoundation.org, 2019.

[25] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *Proceedings of the 2013 Annual ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages & Applications (OOPSLA)*, Indianapolis, IN, Oct. 2013.

[26] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *ACM SigPlan Notices*, vol. 47, no. 6, pp. 193–204, 2012.

[27] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing MAYHEM on Binary Code," in *Proceedings of the 33rd Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[28] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, Hyderabad, India, May 2014.

[29] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and Efficient Malware Detection at the End Host," in *Proceedings of the 18th USENIX Security Symposium (Security)*, Montreal, Canada, Aug. 2009.

[30] H. Lim, "Detecting Malicious Behaviors of Software through Analysis of API Sequence k-grams," *Computer Science and Information Technology*, vol. 4, no. 3, pp. 85–91, 2016.

[31] *Malpedia: Free and Open Malware Reverse Engineering Resource offered by Fraunhofer FKIE*, https://malpedia.caad.fkie.fraunhofer.de, [Accessed: 2019-01-28].

[32] *Malware Archaeology: Malware Discovery, Education, Training, Active Defense, Detection and Response*, https://www.malwarearchaeology.com/analysis, [Accessed: 2019-01-28].

[33] *MITRE ATT&CK Framework: A globally-accessible knowledge base of adversary tactics and techniques based on real-world observations.* https://attack.mitre.org/software/, [Accessed: 2019-04-20].

[34] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers," in *Proceedings of the 36th Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[35] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Computer Security Applications Conference (ACSAC)*, 2016.

[36] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, *Obfuscation benchmarks*, 2016. [Online]. Available: https://github.com/tum-i22/obfuscation-benchmarks.

[37] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, 1976.

[38] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT — a formal system for testing and debugging programs by symbolic execution," *ACM SigPlan Notices*, vol. 10, no. 6, pp. 234–245, 1975.

[39] W. E. Howden, "DISSECT — A symbolic evaluation and program testing system," *IEEE Transactions on Software Engineering*, no. 4, pp. 266–278, 1978.

[40] C. Cadar and D. Engler, "Execution generated test cases: How to make systems code crash itself," in *Proceedings of the International SPIN Workshop on Model Checking of Software*, San Francisco, CA, Aug. 2005.

[41] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: Preliminary assessment," in *Proceedings of the 33th International Conference on Software Engineering (ICSE)*, Honolulu, HI, May 2011.

[42] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, "Selective Symbolic Execution," in *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, Estoril, Portugal, Jun. 2009.

[43] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection*, Springer, 2008, pp. 65–88.

[44] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, Mar. 2012.

[45] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-Force: Force-Executing Binary Programs for Security Applications," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[46] R. Baldoni, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Assisting Malware Analysis with Symbolic Execution: A Case Study," in *Proceedings of the International Conference on Cyber Security Cryptography and Machine Learning (CSCML)*, Israel, Jun. 2017.

[47] B. Yadegari and S. Debray, "Symbolic Execution of Obfuscated Code," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[48] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, Nov. 2009.

[49] W. Cui, M. Peinado, Z. Xu, and E. Chan, "Tracking Rootkit Footprints with a Practical Memory Analysis System," in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[50] J. Rhee, R. Riley, Z. Lin, X. Jiang, and D. Xu, "Data-Centric OS kernel malware characterization," *IEEE Transactions on Information Forensics and Security*, vol. 9, 2014.

[51] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan zee (north) bridge: Mining memory accesses for introspection," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[52] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2010.

[53] A. Slowinska, T. Stancescu, and H. Bos, "Howard: a Dynamic Excavator for Reverse Engineering Data Structures," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2011.

[54] Q. Feng, A. Prakash, H. Yin, and Z. Lin, "Mace: High-coverage and robust memory analysis for commodity operating systems," in *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, 2014.

[55] M. Polino, A. Scorti, F. Maggi, and S. Zanero, "Jackdaw: Towards Automatic Reverse Engineering of Large Datasets of Binaries," in *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Milan, IT, Jul. 2015.

[56] Z. Xu, J. Zhang, G. Gu, and Z. Lin, "Autovac: Automatically extracting system resource constraints and generating vaccines for malware immunization," in *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS)*, 2013.

[57] B. Saltaformaggio, Z. Gu, X. Zhang, and D. Xu, "DSCRETE: Automatic Rendering of Forensic Information from Memory Images via Application Logic Reuse," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[58] B. Saltaformaggio, R. Bhatia, X. Zhang, D. Xu, and G. G. Richard III, "Screen after previous screens: Spatial-temporal recreation of android app displays from memory images," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[59] R. Bhatia, B. Saltaformaggio, S. J. Yang, A. Ali-Gombe, X. Zhang, D. Xu, and G. G. Richard III, ""Tipped Off by Your Memory Allocator": Device-Wide User Activity Sequencing from Android Memory Images," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[60] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "GUITAR: Piecing Together Android App GUIs from Memory Images," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[61] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "VCR: App-Agnostic Recovery of Photographic Evidence from Android Device Memory Images," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[62] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "NoDoze: Combatting Threat Alert Fatigue with Automated Provenance Triage," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[63] W. U. Hassan, A. Bates, and D. Marino, "Tactical provenance analysis for endpoint detection and response systems," in *Proceedings of the 41st Symposium on Security and Privacy (Oakland)*, Online Conference, May 2020.

[64] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: real-time APT detection through correlation of suspicious information flows," in *Proceedings of the 40th Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[65] R. P. Kasturi, Y. Sun, R. Duan, O. Alrawi, E. Asdar, V. Zhu, Y. Kwon, and B. Saltaformaggio, "TARDIS: Rolling back the clock on CMS-targeting cyber attacks," in *Proceedings of the 41st Symposium on Security and Privacy (Oakland)*, Online Conference, May 2020.

[66] Y. Shen and G. Stringhini, "Attack2vec: Leveraging temporal word embeddings to understand the evolution of cyberattacks," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[67] R. Rolles, *Rolfrolles/hexraysdeob*, https://github.com/RolfRolles/HexRaysDeob, Jun. 2018.

## A Appendix: Additional Technical Material

| **Capability Plugin** | **Tracked APIs (Reverse Order)** | **Tracked Parameters** | **Description** |
|---|---|---|---|
| File Exfiltration | Send(socket, buf); Socket(socket); ReadFile(hFile, buf); OpenFile(lpFname) | socket <- Socket(socket); buf <- ReadFile(hFile, buf); hFile <- OpenFile(lpFname) | Exfiltration functionality tracks back from the Send function by tracking Socket creation, file access (OpenFile and ReadFile), and the parameters associated with each API. |
| Code Injection | SetThreadContext(pContext); WriteProcessMemory(hProcess); VirtualAllocEx(hProcess); ZWUnMapViewOfSection(pHandle); GetThreadContext(pContext); CreateProcess(appName) | pContext <- GetThreadContext(pContext); hProcess <- CreateProcess(appName); pHandle <- hProcess | This code injection technique is known as process hollowing. The plugin tracks from SetThreadContext with WriteProcessMemory, and VirtualAllocEx to identify parameter constraints tying back to pContext, hProcess, and pHandle. |
| Dropper | CreateProcess(lpApplicationName); SetFileAttribute(lpFileName); WriteFile (hFile); CreateFile(lpFileName) | hFile <- CreateFile(lpFileName); lpFileName <- lpFileName; lpApplicationName <- lpFileName | This plugin tracks code that writes a file to disk by creating a file handle based on a filename. Then it tracks an attribute modification that sets the property for execution. Then tracks filename and path used in the process creation to make up the dropper capability. |
| Key & Screen Spy | FromHbitmap(hBitmap); SelectObject(hDCDest); CreateCompatibleDC(hDCSource); CreateCompatiableBitmap(hDCSource); GetWindowDC(Whandle); GetDesktopWindow() | hBitmap <- CreateCompatiableBitmap(hDCSource); hDCDest <- CreateCompatiableDC(hDCSource); hDCSource <- GetWindowDC(Whandle); Whandle <- GetDesktopWindow() | This plugin tracks screen capture capability by identifying a handle to bitmap object that constraints on a handle to a device context object. Then constraints the handle to a Windows handle object that is created by referencing the user Window. |
| Persistence | GetFullPathNameA(lpFileName); RegOpenKeyEx(lpSubKey); RegSetValueEx(hKey, lpData) | lpData <- GetFullPathnameA(lpFileName); hKey <- RegOpenKeyEx(lpSubKey:str-match) | This plugin tracks a persistent method that relies on registry keys. Specifically, we track constraints on the file path value set to a key and sub key value by matching for *HKLM*, *HKCU*, *ControlSet*, and *Run* to the registry key handle. |
| Anti-Analysis | InternetGetConnectedStates; GetConnectedProfiles; GetConnectivity; InternetAttemptConnect; OutputDebugString; IsDebuggerPresentPresent; CheckRemoteDebuggerPresent; CreateToolhelp32Snapshot; EnumProcesses; cpuid (instruction) | None specified | This plugin applies no parameter constraints to identify and track anti-analysis capability. Since FORECAST assumes the memory image under analysis is a suspicious or malicious (detected by HIDS), FORECAST simply searches for any invocation of these Windows API functions to track anti-analysis capability. |
| *C&C* Comm. | InternetConnectA(lpszServerName); InternetCheckConnectionA(lpszUrl); IWinHttpRequest::Open(Url) | lpszServerName - IP/Domain regex-match; lpszUrl - IP/Domain regex-match; Url - IP/Domain regex-match | This plugin applies regex match constraint to the parameters of a select network-based APIs to identify and track *C&C* communication. Specifically, the plugin tracks internet routable and valid domain names. |

Table 7: Capability identification and tracking is a modular component of FORECAST. Analysts can build additional capability plugins to help in future investigations by identifying APIs and parameter constraints that make up the capability. The parameter constraints are tracked through data flow analysis and backward slicing.

| Sample | Year Reported | Hash (SHA 256) |
|---|---|---|
| rokrat | 2018 | 4d37f80da97845129debf3244e1f731d2c93a02519f9fdaa059f5f124cf7c26f |
| 7honest | 2016 | 575e6fa02a54b9e3cd5977a66d09cf0e841d6efbe59be334056cf8fe8613194a |
| bokbot | 2019 | 62b7fbffd000a8d747c55260f0b867d09bc4ad19b2b657fb9ee3744c12b87257 |
| AcridRain | 2018 | 7b045eec693e5598b0bb83d21931e9259c8e4825c24ac3d052254e4925738b43 |
| AthenaGo | 2016 | af385c983832273390bb8e72a9617e89becff2809a24a3c76646544375f21d14 |
| AdamLocker | 2016 | 0fb2e4bdd84c3ae8af8fb255ad4f5d093bc10544684bff739ccc985ebd4e64cb |
| Marap | 2018 | 5859a21be4ca9243f6adf70779e6986f518c3748d26c427a385efcd3529d8792 |
| Abaddon | 2015 | 7cfc340ed0bd2af138c4b2b85c19693755a9c9ea798028d1a17d0cfcc61b5a3a |
| ATI | 2016 | b101cd29e18a515753409ae86ce68a4cedbe0d640d385eb24b9bbb69cf8186ae |
| TeslaAgent | 2018 | c2cae82e01d954e3a50feaebcd3f75de7416a851ea855d6f0e8aaac84a507ca3 |
| Andromeda | 2013 | f20355d0e3689bf7e8540c6881cb5299e36c5342a3679dd54d206c4ff4f8b979 |
| AVCrypt | 2018 | 58c7c883785ad27434ca8c9fc20b02885c9c24e884d7f6f1c0cc2908a3e111f2 |
| AveMaria | 2018 | 81043261988c8d85ca005f23c14cf098552960ae4899fc95f54bcae6c5cb35f1 |
| Aveo | 2016 | 9dccfdd2a503ef8614189225bbbac11ee6027590c577afcaada7e042e18625e2 |

Table 8: Malware Samples Used In The Forecasting Evaluation (§4.1).

| Sample Hash | Capture IP | Capability End Address(es) |
|---|---|---|
| f9c6db5331051aa487b706f0616f3287a40a27606bfddc804b3c4684d4203717 | 0x140005057 | 0x140008060 |
| 59b9d061ff78c240e1e0e8135d9be482e0fe788186b6cb940f56c67798a862df | 0x14000515b | 0x140008152 |
| 1eed6b168c2cd7701bf3a2aa6a30cf014cae9bc6ae813ef7356c5c6bc8ad6d18 | 0x1400050e7 | 0x1400080ec |
| 471de9132673ec513b5c7c06a4bc1f67a7e91c6c8c7def55e9e03131ac5fb400 | 0x40109d | 0x401374, 0x77244bb4 |
| 153fb1b9cd5dabffa3d123c4ac91abae46546db7447140df7b4aa1f2d3e8f59e | 0x1400010e6 | 0x1400010f1, 0x77994bb4 |
| ffec8e4a80182eb507489bcabd368d42489bf1ec871542c131df04c068d01a76 | 0x14000221f | 0x140002516, 0x78204bb4 |
| baa0f9e799a3d46ccb04c9d4520a69e58383b2d88aad8746f9214eaa8d3a06f3 | 0x14000f2b1 | 0x140011380, 0x14000f29e |
| ff64690b250faa9b1902b945f543a7b4ff9560cb562c0b18f3798538cc28178c | 0x14000c2ac | 0x14000fa10, 0x14000c299 |
| dc616f2f6b1856454412ea608b96d3d6d7ab719684b6d04f0a79cf9228477d4f | 0x140001408 | 0x1400013fd, 0x140001054 |
| f2f9696ffea5b8cf3c1bf860a3d0704033b7693199cf097367a052144b0c350f | 0x14000b089 | 0x1400234a9, 0x77314bb4 |
| 3025bf51ac1f1571e3f49ee1836d44f0cfd9bfcf6e39731f6fea0ddde33925a1 | 0x1400012d4 | 0x140007690, 0x78714bb4 |
| e82b6a27a1aec373983f189cd422f1eeb336f1f493db341df5d090a4946feae8 | 0x14000159a | 0x140012987, 0x7fefef911fd, 0x77984bb4, 0x14000de82, 0x14000de5f, 0x14000de8d |
| 51c5668f052bbfb4ca9670413a240c8214264839211119543b28f90f86504edc | 0x14000136c | 0x1400043b5 |
| f055f75abb82c9500b3f2cf64f6b546105177599b718304b3fc569e932533087 | 0x14000be19 | 0x14000e360, 0x14000be06 |
| c06b359921a385efbf8ce33bd875a797d89f88c575fe640173429ce5a10b45ae | 0x140001c56 | 0x140002e0e, 0x77ba4bb4 |
| 54b49a2faef8b8a6b8ef9bd96a44575403025e8c422ef8817d8cba6ea0344945 | 0x14000ec06 | 0x140010bc7, 0x14000ebf3 |
| a785bc5be1fd3e9f6997f558a4e613b973769cc43c6e7b738158354b66390d06 | 0x1400012b5 | 0x140004daa, 0x78114bb4 |
| fee18f402375b210fc7b89e29084fb8e478d5ee0f0cdb85d4618d14abb2e5197 | 0x14000faa9 | 0x140011e80, 0x14000fa96 |
| f85abdfa7e8931686bbbb9bb0dd2e12ca10f28b8b1b7be2890eb19023c52232a | 0x1400242b8 | 0x1400242c3, 0x77314bb4 |
| ec72f1af9119754195a77cd890cc9e5ee1e555e9ef89fe2e535ee3e4ce2132cf | 0x140011f10 | 0x140016387, 0x140011efd |
| a5c8d9df73b2ff360d22e879b678d323bbccd81cb9e0ef45cce4aaf4e37c7f27 | 0x140011516 | 0x14001163a, 0x77644bb4 |
| 58f9504b59b40dfbff5e3093af0a39def00b449c499ef3e7c0880ac986575f76 | 0x14000fda9 | 0x140012180, 0x14000fd96 |
| 6130a8c7595f6d9abc3dba157e8bd7596b11c9903296060e52d764a8719d7b84 | 0x140023d5a | 0x77d0a358, 0x140023156, 0x77ba4bb4, 0x77d03e18, 0x140025d80 |
| c9b27cbdc1b4258cd4103b3847e7de9c52985289ce4bd61323d69bf9c1e2a8c0 | 0x1400025bc | 0x14000343c, 0x77874bb4 |
| 1edfad978a9e4beb24c2f51e9cf12424d415f5e9b5292279ac47b9f650495b31 | 0x140001041 | 0x14000174e, 0x140001045, 0x77e1a358, 0x1400016a0, 0x140001437 |
| cab869f98ba3fe1948d2b48fa76fa4767fa7f31e28f3be2b34572ab0c63f942a | 0x14000f8cd | 0x140011ca0, 0x14000f8ba |
| 600845916e82b6de80f9ff1d6a0553ff98bce6f41dc6029343821f095072fcee | 0x14000a6c9 | 0x14001bbb9, 0x77644bb4 |
| c2bda34d3ac4844ea377aa87b115b94019b98919de7d153029865efc969fd46d | 0x140002a5e | 0x76f53e18, 0x140002a62 |
| b59c3d14968a9d7d90baa0df624339aa977dc98e5de1c7f6b71bef23606db769 | 0x14000d20d | 0x14000f1a0, 0x14000d1fa |
| 2dbd5d77540a1470459d74906d1668ae49fb275d834976fae1f31bbe74d8e168 | 0x140003df6 | 0x140010ba0, 0x76cd4bb4 |
| e47b4147f8a51511b087f90ae07a4d0650b17a6ca2be5a7b19ad1c3f058fb15f | 0x4038db | 0x406a8b, 0x7fefbb1580a, 0x405b7a, 0x405b7f, 0x406a7f |
| 6dabcf4ce36360826b381a80a7bd34d0df6612f37528e0086009a87bbc16ee57 | 0x401724 | 0x7fefdaa99e2, 0x7fefdad811e |
| b1dee4864ee0d67afb4889cdb0efe1ea54e1005debeb9ef4b4541848c23750c8 | 0x14000b079 | 0x14001f7e9, 0x77984bb4 |
| f3fb1b8bd66a67e9f5e00895fb1fee886764c1fc65def4b0104eb7408973ee40 | 0x140004750 | 0x778c3e18, 0x77644bb4 |
| b3f91bd440d63ff0b3a28e3fc444714088dc8f30160a6e5f8073594f7d9a6aa6 | 0x1400016ae | 0x1400172b0, 0x7feff5d11fd, 0x77244bb4, 0x140010cdb, 0x140010ce9, 0x140010cb2 |
| d12899958f7adc1be6a3f540f5a25a6ea5eb024dba018d7d3d0a1808df970323 | 0x140004ac2 | 0x140004c00 |
| ae210c336cdfec7f7f523fa5b910981e2896f53184b3863621629e81cc0607ed | 0x14000a747 | 0x78263e18, 0x14000a732 |
| 35b8a197bd6642f62af2b809ba72d8d7cc4ac18879f10cffeb8f2df66db93746 | 0x14000a6cd | 0x14001f869, 0x140005d7e, 0x14000370e |
| c9ee386c3d2b8230d95870ce3391aa8a4890169a0fe021a5562d3735f2466160 | 0x140001238 | 0x140001243 |
| db8caaf17e1e9afa4a64b7e6a57d07a2eb6669edaed70daced81295ea183da9f | 0x14000233c | 0x140002347 |
| 355341b710fe7f121df4c5fcfc32de9da5a5e2003f0869fcbb7a47f92f2471f2 | 0x40369d | 0x403a64, 0x40146e, 0x4014f8 |
| fba0cc427658445f0ca78d6a263c5b9a9714e99e733ffe25ba719c9b39b98664 | 0x14000a685 | 0x140019af9, 0x77ba4bb4 |
| 23c7eee980ca21ac8597bd6eb2147e4bfc1941490db87f276a13146914ea5637 | 0x140003957 | 0x1400075ac, 0x140003945, 0x1400074d6, 0x1400074fe, 0x14000721f |
| 4f998e4290bdf67dc4a1e75ed739eb57defda3c329b6b07f29b3b6c771a8b3ea | 0x1400010ae | 0x1400032d9, 0x77ba4bb4 |
| a238ccc209980719927c777fc9f16866403cb9d58c0c847b9cd92ece0d46e725 | 0x14000226c | 0x14000ab18, 0x14000225a, 0x14000aa42, 0x14000aa6a, 0x14000a79f |
| 17ecabd73e1eb5f7a7f6b35b0c48d3fcf5f73f65aef34993726439d7d27da849 | 0x14000254b | 0x140002556 |
| 5c9e92f6b45b0cb098838e5db6623067396f066704f9c909b31d234bfaf74458 | 0x100005642 | 0x10000c259 |
| 697256960cdded3229b0f2f99b593751d3862774dc7c5cabdbbf769beadd263f | 0x2000032cb | 0x20000da50 |
| c0be7a344a863894890127e61851838037bd9d076423bfc8296cfd6e01d66f6b | 0x14000f939 | 0x140011d10, 0x14000f926 |
| 656ac5ec110c5f8ce68ce1962d6b2cbd47ee6ce20a181c88bb1e5481793f0578 | 0x140001c70 | 0x140001c81, 0x14000133a |

Table 9: Malware Samples And Parameters Used In The Empirical Evaluation (§4.6).