KALI: Scalable Encryption Fingerprinting in Dynamic Malware Traces

Lorenzo De Carli[1] Ruben Torres[2] Gaspar Modelo-Howard[2] Alok Tongaonkar[3] Somesh Jha[4]
[1]Colorado State University [2]Symantec [3]RedLock Inc. [4]University of Wisconsin, Madison

Abstract—**Binary analysis of malware to determine uses of encryption is an important primitive with many critical applications, such as reverse-engineering of malware network communications and decryption of files encrypted by ransomware. The state of the art for encryption fingerprinting in dynamic execution traces, the ALIGOT algorithm—while effective in identifying a range of known ciphers—suffers from significant scalability limitations: in certain cases, even analyzing traces of a few thousands of machine instructions may require prohibitive time/space. In this work, we propose KALI, an enhanced algorithm based on ALIGOT which significantly reduces time/space complexity and increases scalability. Moreover, we propose a technique to focalize the analysis on encryption used for specific purposes, further improving efficiency. Results show that KALI achieves orders of magnitude reduction in execution time and memory utilization compared to ALIGOT, and processes real-world program traces in minutes to hours.**

I. INTRODUCTION

Despite wide-ranging efforts by academia and industry, malware remains one of the most widespread and damaging online threats. Infections can affect individual users and organizations, bringing about significant damage due to financial and reputation loss [1], [2], [3]. Due to the strong economic incentives in creating and distributing malware, these are extremely popular activities, with new samples being observed at rates of millions per year [4]. Given this situation, it is extremely difficult for security operators and developers of anti-malware tools to keep their expertise and techniques up-to-date and effective against the most recent attack strategies.

Developing methods for detecting and counteracting malware infections is complex and involves malware reverse-engineering, which is accomplished by analysis of malware binaries and the network traffic they produce. Such analysis is more art than science, requiring significant expertise and being complicated by defenses deployed by malware, such as virtual machine detection and binary obfuscation. Performing this time-consuming analysis manually on each new malware family is impractical; it is therefore important for the security community to provide tools to automate this process.

A critical aspect of malware functioning is the use of encryption, for purposes such as securing communications between infected machines and command and control (C&C) servers, and scrambling local files as in the recent WannaCry outbreak [3]. It is clear that being able to profile and reverse the use of encryption ciphers is an important analysis primitive which can benefit both malware understanding and detection.

ALIGOT [5], by Calvet et al., is a recently proposed algorithm to fingerprint and reverse symmetric encryption used by malware, with several promising characteristics. First, it is based on dynamic analysis of malware execution traces, which allows it to sidestep hurdles that are traditionally deployed against static binary analysis, such as obfuscation and packing. Also, it works in a largely automated fashion, only requiring access to a malware machine-level instruction trace, and can extract encryption keys and cleartexts. Internally, ALIGOT is based on the insight that loops are recurring structures in cipher implementations, therefore searching execution traces for chains of *dynamic loops*. Once such chains have been identified, each candidate chain is validated by comparing its inputs and outputs against those of known ciphers. At the end of the process, each candidate is either discarded or successfully matched to a known cipher. In the latter case, the algorithm also returns keys, cleartext and ciphertext.

Unfortunately, ALIGOT also has significant limitations which hamper its usefulness in practice. First, the core algorithm for loop detection has time and space complexity which are quadratic in the number of instructions in a trace. While this may appear low, in § V we show that it translates to prohibitive execution times and memory occupations even for traces of a few thousands of instructions. Second, the validation step—which matches candidates to known encryption ciphers—is highly time-consuming, due to the large number of potential interpretations of input and output parameters.

In this paper, we propose KALI, an encryption fingerprinting algorithm which improves ALIGOT to solve its most significant limitations, thus achieving high scalability. In particular, KALI implements a significantly faster loop detection step which internally leverages the Kolpakov/Kucherov algorithm [6] to achieve quasi-linear complexity (KALI stands for **K**olpakov/Kucherov-based **ALI**GOT). Furthermore, it filters candidate encryption functions based on the uses of their output, allowing the analyst to select for example only encryption that affects network data. This operation can significantly limit the number of candidates that must undergo the expensive matching step. Overall, KALI lifts the maximum feasible trace size from tens of thousands of instructions to millions, providing a practical primitive for malware analysts. In previous work, we have used KALI as part of a reverse-engineering pipeline for malware protocols [7], which demonstrates our solution's effectiveness and versatility.

The rest of this paper is organized as follows. In § II we describe and analyze ALIGOT. In § III we introduce KALI and discuss our implementation of the loop detection step, while § IV describes our candidate filtering procedure. § V presents the experimental evaluation, § VI discusses related work, and § VII concludes the paper.

II. Background: the ALIGOT algorithm

ALIGOT, by Calvet et al. [5] is a dynamic program analysis algorithm for heuristically detecting the execution of symmetric encryption algorithms (ciphers) in program instruction traces. The algorithm detects instruction sequences executing encryption ciphers, together with inputs (key, cleartexts) and outputs (ciphertexts). ALIGOT is based on the intuition that symmetric encryption algorithms operate by applying the same computation (encryption) over a stream of data (cleartext) in an iterative fashion. The execution of an encryption primitive therefore presents itself as one or more instruction loops, connected to each other in a chain by input/output relationships. ALIGOT uses a loop detection algorithm to identify dynamic loops, i.e. repeated sequences of instructions, and joins them into chains, each chain representing a candidate encryption instance. Candidates are then matched to known ciphers by comparing their inputs and outputs.

In our operative experience with ALIGOT, we identified several potential pitfalls that may inflate execution time and memory. Most can be solved by simple optimizations, but one issue required significant effort: loop detection takes impractical amount of time/space due to algorithm design. The rest of this section introduces the problem, while § III describes a redesigned loop detection step.

Algorithm 1 Loop detection step for ALIGOT

```
Input: A machine instruction trace $T$
Output: A set $RL$ of all loops in $T$
 1: $RL \leftarrow \emptyset$;
 2: $H \leftarrow \emptyset$; ▷ List of past instructions
 3: for each Instruction $i \in T$ do
 4:     $matched \leftarrow$ true;
 5:     for each $StackOfLoops \in RL$ do
 6:         if MATCH($StackOfLoops$, $i$, $H$) then
 7:             $matched \leftarrow$ true; break;
 8:     if not $matched$ then
 9:         APPEND($H$, $i$);
10:         if there exist other occurrences of $i$ in $H$ then
11:             Create associated loop instances;
12:             Add them to $RL$;
```

A. Overview of ALIGOT loop detection step

Formally, if A_M is the alphabet defined by all possible machine instructions, ALIGOT defines loops as words in the language $L \in \{\alpha^n\beta | \alpha \in {A_M}^+, \beta \in {A_M}^+, n \geq 2, \beta \in Pref(\alpha)\}$. This definition covers all repetitions of a root α, possibly followed by a prefix of α (e.g. the string $ababa$). However, it does not cover nested loops, which are supported with a simple extension. A language L' that includes all loops at nesting depth d can be defined by replacing all loops at nesting depth $d+1$ with loop identifiers, such that loops with the same root α are replaced by the same identifier, and adding all identifiers to the alphabet of L'. Consider for example the string $abababcababc$, which contains two iterations of an outer loop *(ab)+c*. The two iterations of the inner loop have the same root ab, and would therefore be replace by the same identifier l_{ab}, resulting in the string $l_{ab}cl_{ab}c$. We can then define a language L' whose alphabet is $A'_M = \{a, b, c, l_{ab}\}$; the string $l_{ab}cl_{ab}c$ is a valid loop in L'. This suggests a detection procedure based on recursively replacing loops with loop identifiers, which is implemented by ALIGOT.

The loop detection step in ALIGOT uses an approach that detects all loops, including nested ones, in a single pass. The key operations are depicted in Algorithm 1. The instruction trace T is scanned start to end (lines 3-12). If no suitable candidate loops currently exist (line 8), each instruction is considered the potential beginning of a new loop, and a candidate loop structure is created for every previous occurrence of the same instruction (lines 9-12). Candidate loops are then either confirmed or discarded depending on the instructions encountered next (lines 4-7). In particular, for each candidate loop the MATCH procedure on line 6 (omitted for brevity) checks whether the current instruction i corresponds to the next expected instruction in the loop body. If yes, the candidate is retained; if not discarded. Loops that complete at least 2 iterations are confirmed and returned. Furthermore, the algorithm keeps track of nested loops by maintaining a loop stack for each candidate, with each element of the stack corresponding to a nesting level—as soon as the innermost loop terminates, the instructions in the loop are replaced by a loop identifier, and matching continues on the outer loop. We refer the reader to the ALIGOT paper [5] for the details.

B. Complexity of ALIGOT loop detection step

First, we note that—in order to process nested loops—during each step MATCH can recursively call itself up to d times, where d is the maximum loop nesting depth. As d tends to be limited in typical programs, we model MATCH execution time as constant. The main loop (lines 3-12) executes once per instruction in the trace; however, identifying previous occurrences of the current instruction (line 10) requires a scan of the instruction history H, which can grow as large as the trace T itself. As this scan must be performed for every instruction in T, the worst-case time complexity of the algorithm is $O(n^2)$, where $n = |T|$.

As for space complexity, the main auxiliary structure maintained by ALIGOT is RL, which at every step stores both confirmed and candidate loops. A candidate loop is instantiated whenever there exists a previous instance i^{-1} of the current instruction i; creating a candidate loop involves creating a copy of the loop body, i.e. all instructions between i^{-1} and i. The maximum number of concurrent candidates is at least linear in n and storing a candidate requires storing up to $n-1$ instructions. Therefore, ALIGOT's space complexity is $O(n^2)$.

While quadratic complexity may in principle appear acceptable, experimental evaluation—detailed in § V—reveals that in practice it may translate to prohibitively high execution times for traces as small as 10000 instructions. It is therefore necessary to investigate an alternative approach to improve the efficiency of this step, which we discuss in the next section.

III. Scaling ALIGOT: the KALI algorithm

As discussed in § II, ALIGOT's loop detection algorithm has quadratic complexity which results in impractical time and memory consumption. In investigating a potential alternative approach, we observe that the definition of dynamic loop used by ALIGOT—excluding the extension to nested loops—is equivalent to the definition of *maximal repetition* in word combinatorics, which is formalized as follows:

Definition 1 (Period of a word)**.** *Consider a word w of length n consisting of a sequence of characters $a_1...a_n$. The period of w is the smallest positive integer p such that $\forall i$ s.t. $1 \leq i, i+p \leq n, a_i = a_{i+p}$. n/p is called the exponent of w.*

Definition 2 (Maximal repetition)**.** *A maximal repetition in a word $w = a_1...a_n$ is a word $r = a_i...a_j$, with $1 \leq i, j \leq n$, such that:*

1) *If $i > 1$, the period of $a_{i-1}...a_j$ is greater than the period of r.*
2) *If $j < n$, the period of $a_i...a_{j+1}$ is greater than the period of r.*

Note that it is tempting to interpret "maximal repetition" as "longest possible repetition". This interpretation however is misleading, as "maximal" only means that it is not possible to further extend the repetition either on the left or on the right without increasing the repetition's period. Also, maximal repetitions can overlap. An example from [6]: the word *babbababbabba* contains seven maximal repetitions: *babab* (period 2), prefix *babbababbab* (period 5), prefix *babbabba* (period 3), suffix *babbab* (period 3), and *bb* (period 1).

For the purpose of this work, it is interesting to observe that the definition of dynamic loop used by ALIGOT—excluding the extension to nested loops—is equivalent to the definition of maximal repetition given above[1]. Importantly, there exist in literature an approach to find all maximal repetitions in a word of arbitrary length in linear time: the Kolpakov/Kucherov algorithm [6]. This observation informs the design of an improved version of ALIGOT, which we call KALI (**K**olpakov/Kucherov-based **ALI**GOT). In the rest of this section we first introduce the Kolpakov/Kucherov algorithm, and then illustrate how it is used to detect loops in dynamic instruction traces in quasi-linear time.

A. Kolpakov/Kucherov algorithm primer

The Kolpakov/Kucherov algorithm is based on a result by Main [8], which demonstrates how to detect all the *unique leftmost maximal repetitions* in a word—i.e. all the first occurrences of distinct maximal repetitions in the word (assuming that the word is processed left-to-right)—in linear time. By extending this approach, it is possible to detect *all* maximal repetitions in a word in linear time. In particular, the Kolpakov/Kucherov algorithm first uses Main's approach to find all leftmost maximal repetitions, referred in the following as *type-1 repetitions*. It then finds the remaining maximal repetitions, denoted as *type-2 repetitions* (note that type-2 repetitions are duplicates of type-1 repetitions).

[1]Formally, ALIGOT's definition of a loop matches that of *repetition*, and not the more specific one of *maximal repetition*. However, ALIGOT's loop algorithm in practice only returns maximal repetitions.

Algorithm 2 Main steps in the Kolpakov/Kucherov algorithm

```
Input: A word $w$ of arbitrary length
Output: A set $R$ of maximal repetitions in $w$
1: $SA \leftarrow$ COMPUTE_SUFFIX_ARRAY($w$);
2: $SF \leftarrow$ COMPUTE_S_FACTORIZATION($SA$);
3: $R1 \leftarrow$ COMPUTE_TYPE1_REPETITIONS($SF$);
4: $R2 \leftarrow$ COMPUTE_TYPE2_REPETITIONS($SF$, $R1$);
5: $R \leftarrow R1 \cup R2$;
```

1) Detecting type-1 repetitions: Main's algorithm [8] is based on a result which we state without further discussion: *all leftmost maximal repetitions (type-1 repetitions) in a word w lie on the frontier between two s-factors[2] of w.*

Definition 3 (s-factorization)**.** *Consider a word w consisting of a sequence of n symbols $a_1...a_n$. The s-factorization of w is a partition of w in s-factors $u_1...u_k$. Each s-factor u_i is defined inductively:*

1) *If a symbol a immediately following $u_1...u_{i-1}$ does not occur in $u_1...u_{i-1}$, then $u_i = a$.*
2) *Otherwise, u_i is the longest word such that $u_1...u_i$ is a prefix of the word w, and u_i has at least two possibly overlapping occurrences in $u_1...u_i$.*

s-factorization of a word w can be efficiently computed using a *suffix array* for w, which in turn can be constructed using various linear-time algorithms [10]. Main's algorithm first computes such s-factorization for the word of interest, and then uses a set of linear-time auxiliary functions to find all maximal repetitions on the frontiers between factors.

2) Detecting type-2 repetitions: Because of the properties of s-factorization, each factor u_i of a word w is either a novel symbol, or a repeat of a subword encountered previously in the input word w. Therefore, if a maximal repetition occurs inside a factor u_i, it is necessarily a duplicate of a type-1 repetition encountered earlier in the word. In particular, if the earlier occurrence of u_i, which we refer to as t_i, contains a maximal repetition, then the same repetition will occur again shifted to the right, at a distance from the first occurrence equal to the difference between the position of u_i and the position of t_i. In extreme synthesis, step 2 of the Kolpakov/Kucherov algorithm exploits this fact, by keeping track of the earlier occurrence of each factor u_i in a word w, and using this information to retrieve duplicate maximal repetitions. We refer the reader to literature on the subject [6], [9] for further details. Algorithm 2 summarizes the main steps of the Kolpakov/Kucherov algorithm.

[2]Conflicting definitions of s-factorization seem to exist in literature. Kolpakov and Kucherov define it as equivalent of Lempel-Ziv factorization [6], however in [9] the same authors call the factorization given here *s-factorization*, and give a different definition of Lempel-Ziv factorization. Moreover, Ohlebusch and Gog's definition of Lempel-Ziv factorization [10] is equivalent to Lothaire's (and ours) definition of s-factorization. To avoid confusion, we refer to the operation solely as *s-factorization*.

B. *The* KALI *algorithm*

KALI replaces ALIGOT's loop detection step (ref. § II) with one based on the Kolpakov/Kucherov algorithm described above. The KALI loop detection step is given in Algorithm 3. At its core, KALI detects loops (line 4) by running the Kolpakov/Kucherov algorithm (function FIND_MAXIMAL_REPETITIONS, defined in Algorithm 2) on the input instruction trace T, and collecting all the detected maximal repetitions. However, this is not sufficient because the Kolpakov/Kucherov algorithm per se cannot detect nested loops. Furthermore, ambiguities can arise with certain sequences of instructions. We deal with both issues using post-processing steps.

Algorithm 3 Loop detection step for KALI

```
Input: A machine instruction trace $T$
Output: A set $R$ of all loops in $T$
 1: $R \leftarrow \emptyset$;
 2: do
 3:     ▷ Repetitions in L are sorted by start, end position
 4:     $L \leftarrow$ FIND_MAXIMAL_REPETITIONS($T$);
 5:     $A \leftarrow \emptyset$;
 6:     for each Repetition $r \in L$ do;
 7:         $accept \leftarrow$ true;
 8:         for each Repetition $r' \in A$ do
 9:             if overlap$(r, r')$ then
10:                 $accept \leftarrow$ false; break;
11:         if $accept =$ true then
12:             $A = A \cup \{r\}$;
13:     for each Repetition $r \in A$ do
14:         $T \leftarrow$ REPLACE_WITH_ID$(T, r)$;
15:     $R \leftarrow R \cup A$;
16: while $A \neq \emptyset$;
```

1) Nested loop detection: Our approach performs multiple passes over the input trace T, each detecting one level of nested loops. This is described in Algorithm 3, lines 2-16. Ignoring lines 5-12 for now, lines 13-14 iterate over all accepted maximal repetitions detected in one pass. The auxiliary function REPLACE_WITH_ID takes each detected loop and replaces its body in the trace by a loop identifier. As an example, consider again the string $abababcababc$ consisting of two iterations of the outer loop *(ab)+c*. We would first identify all innermost loops $(ab)+$ and replace them by an identifier, lending to the string l_1cl_1c. We would then run the Kolpakov/Kucherov algorithm again, this time identifying the outer loop l_2. Our approach iterates until no further loops are detected in one pass, in general performing $d+1$ passes, where d is the maximum loop nesting depth in the trace.

2) Loop filtering: An instruction trace may contain ambiguous sequences of instructions, that may be interpreted as containing different sets of potentially overlapping loops. As in a valid program execution loops cannot overlap, such ambiguities must be resolved by selecting some of the overlapping loops, and discarding the others. One case of particular interest is that of nested loops, that can create artifacts in the trace. For example, consider the string $ababac, abababac, ababababac$ (with commas inserted for clarity and not part of the string), representing an outer loop with body $(ab)+ac$, where the inner loop with body ab repeats a variable number of times at each iteration. The Kolpakov/Kucherov algorithm will correctly locate all the instances of the inner loop $(ab)+$ but it will also identify the apparent loop $(ababacab)\{2\}$. The original ALIGOT loop detection algorithm resolves such ambiguities by greedily accepting loops; i.e. as soon as a loop is identified it is stored, and the instructions which are part of its body become unavailable for constructing other loops. This is a reasonable heuristic (although without any guarantee of correctness), and in the situation above it will greedily accept the first, shorter loop, and it will not generate the fake loop $(ababacab)\{2\}$. In our approach, we simulate this strategy by post-processing the list of maximal repetitions in this way:

1) Sort list of maximal repetitions by start position, use end position as sorting criterion to resolve ties.
2) Traverse the sorted list of maximal repetitions. At each position, examine the current repetition:
   - If the current repetition does not overlap with any accepted repetition, accept it,
   - If the current repetition does overlap with one or more accepted repetitions, discard it.

Sorting repetitions by start position and end position guarantees that, in case multiple loops overlap, the one occurring earlier is accepted, which is the same heuristic selection strategy implicitly adopted by ALIGOT. In Algorithm 3, the filtering operation is implemented in lines 5-12; note that FIND_MAXIMAL_REPETITIONS internally already sorts the repetitions, as this is necessary for finding type-2 repetitions.

C. *Complexity of* KALI

Detecting maximal repetition is accomplished in time linear in the length n of the trace T. Checking overlap between repetitions has complexity $O(r^2)$ where r is the number of maximal repetitions in T. Note that both operations must be repeated d times, where d is the maximum nesting depth; however as discussed in § II, we treat d as a small multiplicative constant. Therefore, the overall time complexity of our loop detection approach is $O(n + r^2)$. In contrast, ALIGOT's original loop detection step has complexity $O(n^2)$; note that in realistic program traces $n >> r$. In practice, we observed that KALI lifts the maximum trace size that is practical to analyze with ALIGOT by orders of magnitude, as evaluated in § V.

In terms of space complexity, the main working data structure kept by KALI—besides the trace T itself—is the list of loops R; however, differently from ALIGOT, KALI only stores confirmed loops, so the memory occupation of this structure is $O(r)$ and negligible. As for FIND_MAXIMAL_REPETITIONS, the most expensive operation in terms of memory occupation is the s-factorization; we use an algorithm by Ohlebusch and Gog whose space complexity is $O(n)$ [10].

IV. Classifying encryption uses

Besides the loop detection step described in § III, in Kali we deploy a second optimization which decreases the overall number of candidate encryption instances that are matched against known ciphers. This is motivated by two observations. The first is that matching each candidate to a known cipher is a time-consuming operation. At high level, the set of inputs $IN(G)$ to an encryption candidate G is passed to each element k in a set K of known ciphers. If the output of any known cipher $k \in K$ is equal to the encryption candidate's output $OUT(G)$, then the candidate G is accepted and associated to the cipher. However, this approach faces scalability issues as there are many different possible mappings from the low-level inputs of an encryption instance (contents of individual memory locations and machine registers) to the high-level inputs of a cipher algorithm (variables with abstract types such as `string key`, `byte[] cleartext`). A similar problem is faced when dealing with the outputs of the encryption candidate. ALIGOT tackles the instance-to-cipher matching problem by considering many possible mappings from the low-level parameters of an encryption instance to the high-level parameters of a cipher implementation. This operation is computationally intensive because it requires executing all ciphers supported by ALIGOT on every possible interpretation of the input and output parameters, of which there are many. For example, the matching operation on the malware traces detailed in § V requires 2.33 seconds/candidate. The second observation is that in practice not all uses of encryption may be of interest to the analyst. For example, if the analysis is targeted at decrypting the malware's network communications (as previously proposed in [7]), only encryption candidates whose output is sent on the network should be considered for matching. Similarly, if the analysis targets a ransomware the analyst may be only interested in encryption of local files.

Based on these two observations, we further modify the ALIGOT algorithm to enable considering only candidate encryption instances whose output is used for a particular purpose, thus achieving a reduction (quantified in § V) in the number of candidates that must undergo the expensive matching operation. In particular, we found that an approach that works well is to infer such purpose from the type of system calls that make use of the encrypted output. This requires an overhaul of the way ALIGOT proceeds in analyzing the traces, which we outline in the rest of this section.

A. System call tracing

First, to deploy our optimization scheme the sandbox recording the malware execution traces must be modified to also extract the system calls of interest, along with each call's input parameters. Our current implementation supports selecting system calls used to send data over the network. In particular, we support the following calls from the Windows WSA API: `WSASend`, `WSASendTo`, `send`, `sendto`, `TransmitFile`, and `InternetWriteFile`. It should be noted that modern OS APIs may offer many different ways to send data over the network; although this set of system calls is sufficient for the malware we evaluated we make no claim that it is complete.

The overhead of capturing these calls is negligible compared to that of generating the instruction trace; an analysis of our execution traces from three malware families shows that one system call of interest is issued every ∼140K instructions.

Algorithm 4 High-level operations in Kali

```
Input: An instruction trace $T$
Input: A location $l$ in $T$
Output: A set $E$ of encryption instances
1:  $T' \leftarrow T_1, ..., T_l$; ▷ Truncate trace
2:  $R \leftarrow$ DETECT_LOOPS($T'$); ▷ ref. Algorithm 3
3:  $C \leftarrow$ BUILD_LOOP_CHAINS($R$);
4:  for each Loop chain $G \in C$ do
5:      $Outputs \leftarrow OUT(G)$;
6:      for each Instruction $i \in T'_{LAST(G)+1} ... T'_l$ do
7:          $Outputs \leftarrow Outputs \setminus DEF(i)$;
8:      if $Outputs \cap USE(T'_l) = \emptyset$ then
9:          $C \leftarrow C \setminus \{G\}$;
10: $E \leftarrow \emptyset$;
11: for each Remaining loop chain $G \in C$ do
12:     $k \leftarrow$ MATCH($G$); ▷ match G against known ciphers
13:     if $k \neq \bot$ then ▷ Check if a match has been found
14:         $E \leftarrow E \cup (G, K)$;
```

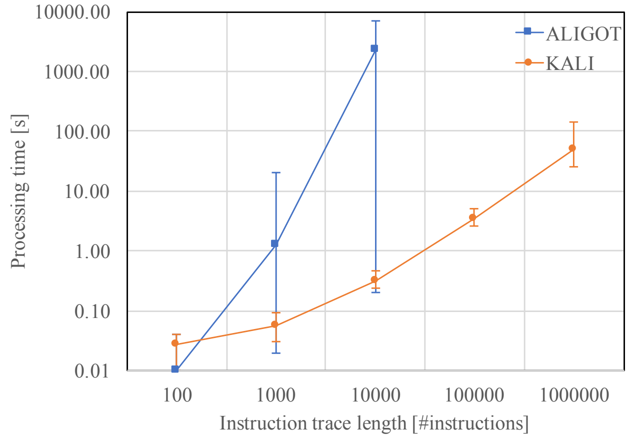
B. Encryption candidate filtering

While ALIGOT receives as input an instruction trace T which it analyzes in its entirety, Kali receives as input an instruction trace T—including both machine instructions and system calls of interest—and the location l of a system call within T. It then truncates the trace at that system call, and only processes candidate encryption instances whose output is used by the system call. The locations of system calls of interests are obtained by a simple pre-processing step.

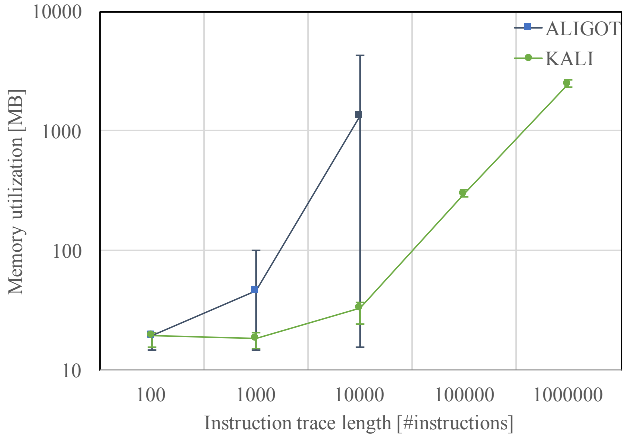
Algorithm 4 describes the overall Kali algorithm. It makes use of a few simple auxiliary functions: $DEF(i)$ ($USE(i)$) returns the set of memory locations written (read) by instruction i. $LAST(G)$ returns the location of the last instruction in a loop chain G within T. $OUT(G)$ returns the output parameters of a loop chain G. Informally, the output of a loop chain G is defined as all memory locations written by loops in G which are not reused as input by other loops in G.

Lines 1-3 detect loops, assemble them in chains, and in the process compute input and output parameters for each chain. Then, lines 4-9 verify whether the output of each candidate chain overlaps with the input of the system call of interest. Care must be taken to ensure no other instruction overwrites the candidate's output before it is used by the system call (lines 5-7). Lines 10-14 match the candidate's inputs and outputs against known encryption ciphers, returning positive matches.

Note that in our current implementation the Kali algorithm must be run separately for every system call of interest; however the set of encryption candidates and their outputs can be memoized so that they are only computed once.



(a) Execution times



(b) Memory utilization

Fig. 1: Comparison of execution times and memory utilization for ALIGOT and KALI

Malware	**Execution time [s]**	**Max trace length [insn]**	**Cipher**
Sality	872	17770293	RC4
Ramnit	414	2355323	RC4
ZeroAccess	1617	9094518	RC4

TABLE I: Details of the traces generated by each malware

V. EVALUATION

In this section, we compare time and memory occupation for ALIGOT's and KALI's loop detection steps, using as inputs instruction traces of various sizes from three different malware families. Results show that the optimizations described in § III are necessary in order to process traces of realistic size. Overall, KALI decreases processing times by up to 4 orders of magnitude and memory occupation by up to 2 orders of magnitude in in the loop detection step.

Furthermore, we analyze an implementation of KALI's classification step which can isolate encryption executions used to generate networking data; this filtering step results in up to a 70% reduction in the number of candidate encryption instances that must be further analyzed.

A. *Implementation and datasets*

The evaluation of ALIGOT is based on the code published by its authors [11]. Our implementation of KALI consists of a fork of ALIGOT's Python code which implements our improvements; we implemented the Kolpakov/Kucherov algorithm in a separated C++ codebase which is then called by the KALI code. Our implementation of Kolpakov/Kucherov internally uses Angelos Molfetas's code for s-factorization [12], however we replace the suffix array construction step with one due to Yuta Mori [13], which is significant more space-efficient for large alphabet sizes.

The instruction traces used to evaluate ALIGOT and KALI were generated by running binaries from three different malware families (Sality [14], Ramnit [15] and ZeroAccess [1]) in a sandbox. The sandbox is based on the Cuckoo software [16], which we modified in order to use PIN [17] to capture machine instruction traces. We only traced instructions from malware binary (or injected process), and not library code. Table I details the input traces specifying for how long each malware binary executed, the maximum trace length across all threads, and the cipher being used. We run all the experiments in this section on a MacBook Pro laptop equipped with 16GB of RAM and a 4-core Intel Core I7 processor running at 2.9GHz.

B. *Evaluation of loop detection step*

This section compares the performance of ALIGOT and KALI loop detection steps, focusing in particular on scalability. In order to do so we extracted random subtraces of various lengths from the traces detailed in Table I and we fed them as input to both algorithms, measuring execution times and memory occupation. In particular, we generated subtraces of length 100, 1000, 10000, 100000 and 1000000. For each of the first two cases, our evaluation set consist of 1000 different traces; for the remaining three, we used smaller sets of 100 traces each, in order to bound experiment time. For the same reason, we cap execution time for each run to two hours.

Results concerning execution times are presented in Figure 1(a), while results on memory utilization are presented in Figure 1(b) (note the use of log scale on all axes). Results for ALIGOT on traces of length 100000 and 1000000 are missing because the algorithm times out in most runs—in particular, of the 100 traces of length 100000 that we considered, only 5 were completed within the 2 hours time bound. Further analysis of the timed-out runs reveals an average completion rate (in terms of instructions processed at termination) of 20%.

Analysis of the 100-, 1000- and 10000-instruction runs evidences two significant issues affecting ALIGOT. The first is a superlinear increase in running time and memory occupation depending on trace size, which confirms the results of the complexity analysis presented in § II. The second is a high variability in processing times, which are strongly dependent on the structure of individual traces. Briefly, this depends on the fact that traces containing high density of loops enable

Malware	#instructions (total)	#insn (truncated traces)	#candidates (truncated traces)	#candidates (after filtering)	Avg processing time
Sality	68M	880079	2136	1696	2.30s
Ramnit	13M	119310	341	110	2.82s
ZeroAccess	19M	8504	4	3	<0.01s

TABLE II: Impact of trace truncation and candidate filtering

the algorithm to quickly confirm pending loop candidates and shorten the instruction history that must be considered. In contrast, traces containing long straight-line code sequences force the algorithm to maintain a long history and instantiate large numbers of bogus candidate loops. A significant implication is that malware can force worst-case complexity (making analysis impractical) by executing the right instruction mix.

Results for KALI show reduced time and memory requirements compared to ALIGOT on large traces: the algorithm completes in under three minutes in all cases, with a maximum memory consumption of 2.7GB. In particular, average completion times (memory utilization) for ALIGOT and KALI on the 10000-instruction case are respectively 2337s (1331MB) and 0.32s (32MB). ALIGOT is slightly faster on very short traces, likely due to the slow file-based interface between the KALI Python code and our C++ Kolpakov/Kucherov implementation, whose overhead is high for small inputs (as future work we plan to upgrade the implementation to a more efficient Python wrapper). Another desirable result of KALI is that the particular choice of trace has only a small effect on running times, and negligible effect on memory consumption. Furthermore, excluding the 100- and 1000-instruction trace cases (where the overhead discussed above is likely to dominate), both time and memory show a clear linear dependence on trace size. Overall, these results encourage us to conclude that KALI's redesigned loop detection step succeeds in improving efficiency and scalability of ALIGOT.

C. *Evaluation of candidate filtering strategy*

In this section, we evaluate the impact of the candidate filtering strategy described in § IV on the efficiency of the encryption analysis process. Such impact is twofold. First, the fact that traces are truncated at the point where a system call of interest occur (line 1 in Algorithm 4) implies a reduced workload on the loop analysis process (lines 2-9) which generates encryption candidates (i.e. loop detection and loop chain construction). Second, candidates generated from the truncated traces are further filtered, and only candidates whose output is received by system calls of interest is retained.

To evaluate both effects, we performed loop detection and loop chain construction both with and without filtering. The filtering step used in this experiment determines the set of system calls as follows: first, it finds all instances of network system calls within the traces (ref. to § IV for a list). Then, it retains network system calls whose output matches the content of network packets generated by the malware.

Results are shown in Table II. For each malware, it presents the overall number of instructions across all traces and the number of instructions actually analyzed due to truncation (columns 2 and 3). It also shows the number of encryption candidates in the truncated traces (column 4) and the number of candidates retained after filtering (column 5). For the candidates that are processed, we show the average per-candidate processing time in the matching step (column 6). While in our evaluation trace, truncation brings an orders of magnitude reduction in the number of instruction that must be processed, we caution against generalizing this observation. In fact, it is likely to depend on the fact that the malware samples being considered attempt to contact their botnet shortly after infection and other malware families may behave differently. Candidate filtering brings a significant reduction of the number of candidates—between 20% and 71% in our traces. Overall, results show the filtering step to be beneficial, since processing of each candidate may require up to several seconds.

Malware	Loop detection	Loop chain building	Matching
Sality	11.98s/198MB	478s/2124MB	4064s/364MB
Ramnit	2.00s/284MB	415s/1047MB	310s/1124MB
ZeroAccess	0.19s/145MB	2.97ss/569MB	0.02s/15MB

TABLE III: Time and memory requirement for KALI's steps

D. *Aggregate time and memory requirements of* KALI

This section profiles the overall execution times of KALI on our input traces, breaking them down by individual steps. Results are summarized in Table III; each row refers to the analysis of traces generated by a different malware family. Individual columns show execution time and highest memory consumption for the main steps in the algorithm: loop detection (line 2 in Algorithm 4), loop chain construction and filtering (lines 3-9) and candidate matching (lines 10-14). As can be seen, KALI completes the analysis of real-world malware traces in practical time and memory. Results also confirm that—without KALI's optimizations—loop detection would constitute a significant bottleneck in the algorithm.

VI. RELATED WORK

A. *Encryption reverse engineering*

Works in this category are perhaps the closest to KALI. They attempt to identify cleartext corresponding to malware-encrypted content. Lutz [18] uses dynamic analysis to identify the location of decrypted buffers in memory. It makes no attempt to identify the specific cipher being used, and assumes encryption algorithms are implemented as discrete functions, which may not be true in obfuscated code. Dispatcher [19] and ReFormat [20] use a similar approach and like the work by Lutz do not determine the ciphers being used. Lestringant et al. [21] propose an approach based on static analysis, which does not target obfuscated malware code. Zhao et al. [22] use dynamic taint analysis to identify procedures performing encryption; however the boundaries of such procedures may not be available in obfuscated code. CryptoHunt [23] is a approach partly based on ALIGOT, but uses symbolic loop mapping to map encryption candidates to ciphers, leading to a potentially more precise matching step. As it reuses ALIGOT's loop detection, KALI optimizations apply to this work too.

B. Encrypted content labeling

The goal of these techniques is to label encrypted content for the purpose of associating network activity with specific malware families. They either require previous understanding of the cipher being used, or ignore it. ProVex [24] identifies encrypted malware traffic by applying a battery of decryption algorithms to network messages, and then performing a probabilistic analysis on decrypted content. CoCospot [25] works similarly but use features which are not affected by encryption. Cisco appliances can identify malware traffic regardless of encryption [26], using non-payload-related features.

C. Identification of encryption activities

The purpose of works in this category is to identify and block encryption-related program behavior which is frequently associated with ransomware infections, using high-level behavioral features. Their goal is orthogonal to ours of precisely characterizing malware uses of encryption. Unveil [27] generates fake honeypot users whose files are then monitored to identify ransomware-like activity. Similarly, CryptoDrop [28] and HelDroid [29] identify and block ransomware by using a set of indicators such as file manipulations and display of ransom requests. Yang et al. [30] propose an approach which uses a combination of static and dynamic analysis to identify Android ransomware. Finally, PayBreak [31] stores keys used by ransomware, enabling decryption of locked files.

VII. CONCLUSIONS

This paper presented KALI, an algorithm for detection of cryptographic functions in obfuscated malware code. KALI is based on the popular ALIGOT algorithm, but deploys a number of optimizations which significantly decrease the execution time, enabling its application to large instruction traces which are typically generated by sandboxed malware executions. As such, it can provide a powerful and practical tool to the analyst, potentially sparing the complex and time-consuming manual process of encryption reverse-engineering.

REFERENCES

[1] J. Wyke, "The ZeroAccess Botnet - Mining and Fraud for Massive Financial Gain," Sophos, Tech. Rep., Sep. 2012.

[2] L. Kessem, "Ramnit Rears Its Ugly Head Again, Targets Major UK Banks," Aug. 2016. [Online]. Available: https://securityintelligence.com/ramnit-rears-its-ugly-head-again-targets-major-uk-banks/

[3] L. H. Newman, "The Ransomware Meltdown Experts Warned About Is Here," May 2017. [Online]. Available: https://www.wired.com/2017/05/ransomware-meltdown-experts-warned/

[4] Symantec, "Internet Security Threat Report," Tech. Rep., Apr. 2017.

[5] J. Calvet, J. M. Fernandez, and J.-Y. Marion, "Aligot: Cryptographic Function Identification in Obfuscated Binary Programs," in *CCS*, 2012.

[6] R. Kolpakov and G. Kucherov, "Finding maximal repetitions in a word in linear time," in *FOCS*, 1999.

[7] L. De Carli, R. Torres, G. Modelo-Howard, A. Tongaonkar, and S. Jha, "Botnet Protocol Inference in the Presence of Encrypted Traffic," in *INFOCOM*, 2017.

[8] M. G. Main, "Detecting Leftmost Maximal Periodicities," *Discrete Appl. Math.*, vol. 25, no. 1-2, pp. 145–153, Sep. 1989.

[9] M. Lothaire, *Applied combinatorics on words*, ser. Encyclopedia of mathematics and its applications. Cambridge, UK ; New York: Cambridge University Press, 2005, no. v. 104.

[10] E. Ohlebusch and S. Gog, "Lempel-Ziv Factorization Revisited," in *Combinatorial Pattern Matching*, ser. Lecture Notes in Computer Science, R. Giancarlo and G. Manzini, Eds. Springer Berlin Heidelberg, Jun. 2011, no. 6661, pp. 15–26.

[11] J. Calvet, "Aligot source code," Aug. 2015. [Online]. Available: https://code.google.com/p/aligot/

[12] A. Molfetas, "dstsc: Data Structures and related algorithms (C/C++ implementation)," May 2017. [Online]. Available: https://github.com/Ichindar/dstsc

[13] Y. Mori, "SAIS library," Sep. 2016. [Online]. Available: https://sites.google.com/site/yuta256/sais

[14] N. Falliere, "Sality: Story of a peer-to-peer viral network," Symantec, Tech. Rep., Jul. 2011.

[15] Symantec Security Response, "W.32 Ramnit Analysis," Symantec, Tech. Rep., Feb. 2015.

[16] Cuckoo Foundation, "Cuckoo Sandbox," Oct. 2016. [Online]. Available: http://cuckoosandbox.org/

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *PLDI*, 2005.

[18] N. Lutz, "Towards revealing attackers intent by automatically decrypting network traffic," Ph.D. dissertation, ETH Zürich, 2008.

[19] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering," in *CCS*, 2009.

[20] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, "ReFormat: Automatic reverse engineering of encrypted messages," in *ESORICS*, 2009.

[21] P. Lestringant, F. Guihery, and P.-A. Fouque, "Automated Identification of Cryptographic Primitives in Binary Code with Data Flow Graph Isomorphism," in *ASIA CCS*, 2015.

[22] R. Zhao, D. Gu, J. Li, and Y. Zhang, "Automatic detection and analysis of encrypted messages in malware," in *International Conference on Information Security and Cryptology*, 2013.

[23] D. Xu, J. Ming, and D. Wu, "Cryptographic Function Detection in Obfuscated Binaries via Bit-precise Symbolic Loop Mapping," in *IEEE S&P*, 2017.

[24] C. Rossow and C. J. Dietrich, "Provex: Detecting botnets with encrypted command and control channels," in *DIMVA*, 2013.

[25] C. J. Dietrich, C. Rossow, and N. Pohlmann, "CoCoSpot: Clustering and recognizing botnet command and control channels using traffic analysis," *Computer Networks*, vol. 57, no. 2, pp. 475–486, Feb. 2013.

[26] J. Deign, "Seeing threats hidden in encrypted traffic," Jun. 2017. [Online]. Available: https://newsroom.cisco.com/feature-content?type=webcontent&articleId=1853370

[27] A. Kharraz, S. Arshad, C. Mulliner, W. K. Robertson, and E. Kirda, "Unveil: A large-scale, automated approach to detecting ransomware." in *USENIX Security Symposium*, 2016.

[28] N. Scaife, H. Carter, P. Traynor, and K. R. B. Butler, "CryptoLock (and Drop It): Stopping Ransomware Attacks on User Data," in *ICDCS*, 2016.

[29] N. Andronio, S. Zanero, and F. Maggi, "HelDroid: Dissecting and Detecting Mobile Ransomware," in *RAID*, 2015.

[30] T. Yang, Y. Yang, K. Qian, D. C.-T. Lo, Y. Qian, and L. Tao, "Automated Detection and Analysis for Android Ransomware," in *HPCC*, 2015.

[31] E. Kolodenker, W. Koch, G. Stringhini, and M. Egele, "PayBreak: Defense Against Cryptographic Ransomware," in *ASIA CCS*, 2017.