Botnet Protocol Inference in the Presence of Encrypted Traffic*

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*—Network protocol reverse engineering of botnet command and control (C&C) is a challenging task, which requires various manual steps and a significant amount of domain knowledge. Furthermore, most of today's C&C protocols are encrypted, which prevents any analysis on the traffic without first discovering the encryption algorithm and key. To address these challenges, we present an end-to-end system for automatically discovering the encryption algorithm and keys, generating a protocol specification for the C&C traffic, and crafting effective network signatures. In order to infer the encryption algorithm and key, we enhance state-of-the-art techniques to extract this information using lightweight binary analysis. In order to generate protocol specifications we infer field types purely by analyzing network traffic. We evaluate our approach on three prominent malware families: Sality, ZeroAccess and Ramnit. Our results are encouraging: the approach decrypts all three protocols, detects 97% of fields whose semantics are supported, and infers specifications that correctly align with real protocol specifications.**

I. Introduction

Modern malware increasingly instantiates botnets: network of compromised computers that perform a diverse set of malicious activities. Usually motivated by financial gain, botnets incorporate a wide range of functionalities, such as launching DDoS and spam campaigns, stealing personal information, and committing e-commerce fraud. The large size of many botnets [1], [2] make them an extremely effective attack tool.

A crucial aspect of botnets, given their distributed nature, is network communication. Bots on infected machines routinely "phone home" to botnet servers, receive instructions, and perform data exfiltration. Understanding the semantics of botnet communications allows the analyst to quickly gain insight to the malware's mechanisms, and to generate *network signatures* to detect malicious traffic. Understanding malware communications can also enable remediation actions and forensics.

In order to understand bot network behavior, analysts use a variety of techniques such as manual traffic collection/inspection and sandboxed execution of malware. Increasing use of encrypted communication also imposes manual debugging of malware binaries to analyze and reverse encryption. These procedures are time-consuming, error-prone and unable to cope with the rates at which new malware appears. Limited, manually-inferred protocol understanding may also result in ineffective signatures, which are excessively specific to the available traffic samples, and can be easily circumvented.

In this paper, we propose a novel technique for automatic inference of malware network protocol specifications, given samples of a malware's communications and the malware binary. We focus on malware *Command & Control (C&C)* protocols, used by botmasters to control malware-infected hosts and to execute various malicious activities. Such communications tend to be based on custom binary formats [3], making it possible to specialize the analysis to this domain. Also, each C&C protocol is specific to its malware family—providing the advantages of constituting a reliable fingerprint, and giving insight into the malware structure and intent.

Reverse-engineering a malware protocol is extremely difficult: messages are oftentimes ambiguous, and may contain errors or purposely-injected noise. Our approach casts it as a *type inferencing* problem [4]: we assume that each message consists of a sequence of binary fields, and we define a *type system* describing all possible field types. We then run a novel type inferencing algorithm to infer message structure.

Another significant issue is that—given the wide adoption of encryption in C&C network traffic [3]—decryption is required before inference can be accomplished. We therefore also propose a system to extract C&C encryption keys by applying dynamic analysis on the malware binary. Our approach extends existing techniques (such as [5]), focusing their application on uses of encryption involving network data.

Our approach can significantly alleviate the analysis burden on human analysts, facilitating timely deployment of countermeasures against new malware. Also, a signature generation step based on our specifications can leverage rich semantic understanding, leading to signatures that are resilient to incremental changes in the C&C protocol. Finally, differently from previous works we strive to provide a practical workflow for the analyst, by: (i) allowing to leverage both traffic generated in a controlled environment and samples captured in the wild, and (ii) discovering rich field types that provide actionable material (e.g. downloaded executables).

We evaluated our approach on three different malware protocols. Results demonstrate that our prototype correctly infers details of the encryption used in each protocol, enabling decryption. Furthermore, it detects 97% of fields whose type is supported by our approach, with negligible false positives. Finally, an evaluation of signatures based on the generated specification shows 98% sample coverage/no false positives.

In summary, this paper contributes: *(i) a technique for rich malware protocol inference requiring only decrypted network traffic*, and *(ii) a practical, scalable solution to infer details of encryption used by malware in network protocols, leveraging state-of-the art encryption analysis techniques.*

* Supported in part by NSF grant CNS-1228782. Emails: ldecarli@colostate.edu, {ruben_torresguerra, gaspar_modelohoward}@symantec.com, alok@redlock.io, jha@cs.wisc.edu.

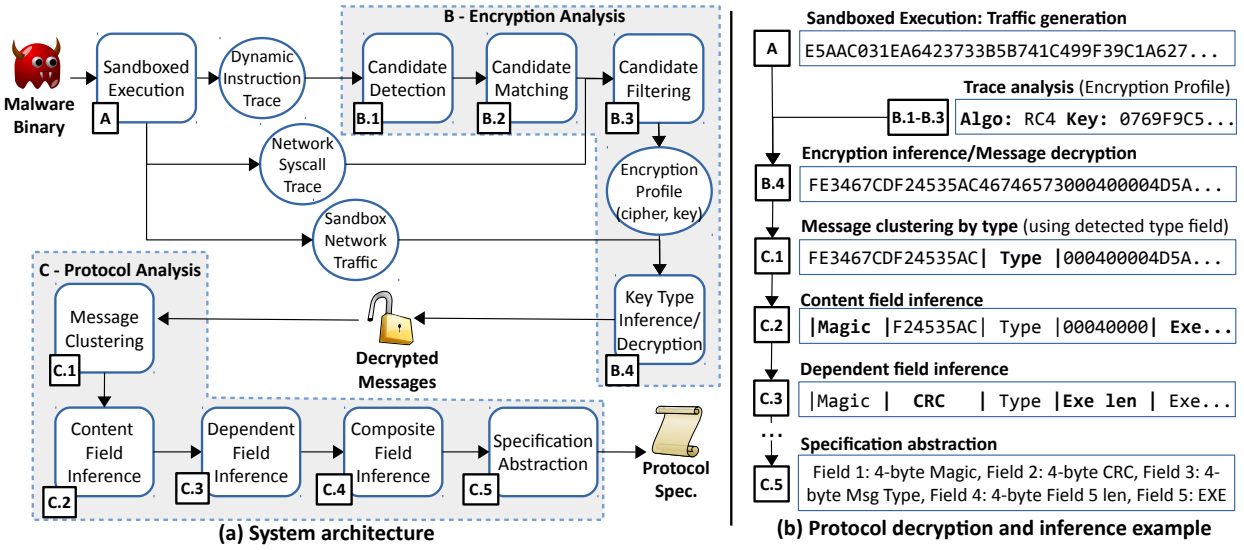


Fig. 1: Outline of the proposed system and example of operation

II. Overview

We decompose the problem of reverse-engineering C&C protocols into: (1) C&C traffic decryption, using dynamic analysis to extract encryption keys from malware executions, and (2) automatic derivation of protocol specification via type inference over decrypted C&C traffic. This section outlines both steps, using as an example a message format from the C&C protocol of the ZeroAccess malware [6] (details of each step are then given in § III). Such message format is used by ZeroAccess to download executable payloads to an infected machine. It consists of a sequence of binary fields: (i) 4-byte magic value that identifies the protocol, (ii) 4-byte message CRC, (iii) 4-byte message type, (iv) 4-byte payload length, and (v) payload (executable file). Each message is fully encrypted using the RC4 cipher and a static key. The overall system architecture is depicted in Fig. 1(a). Fig. 1(b) highlights the output of each step using an example message.

a) Encryption analysis: In step **A**, the malware of interest is executed in a sandbox which traces machine instructions, network system calls, and generated/received network packets. The instruction trace is then processed by a sequence of analysis steps. Step **B.1** pre-selects candidate instruction sequences that behave like encryption functions. Step **B.2** performs a more detailed analysis of each candidate, either mapping it to a specific known encryption cipher, or discarding it. Each surviving candidate represents the dynamic execution of an encryption function. Step **B.3** matches the output of each candidate to the input of network system calls, retaining only candidates whose output is sent on the network (the other candidates likely represent non-C&C related events—e.g. encryption of local files). Step **B.4** maps the retained uses of encryption to the payload of C&C packets emitted during sandbox execution. It then heuristically infers if the encryption key is either static or derived from the payload: if either condition is true, decryption of any malware messages (not just sandbox-generated ones) becomes possible. At the end of this step, all messages in the dataset get decrypted.

Integrating additional samples: Once our approach has learned decryption information, it can decrypt any protocol message. This enables the analyst to feed additional C&C messages from external sources (honeypots, etc.) into the protocol analysis stage, potentially inferring a richer specification.

b) Protocol analysis: Most C&C protocols define multiple message types for different purposes. As each type has a distinct structure, it is crucial that messages are clustered by type, so each type can be analyzed separately. Step **C.1** detects fields specifying the *message type*, and clusters messages based on the values of those fields. Step **C.2** identifies *content fields*, i.e. fields whose structure is self-evident and *independent from the context* (the rest of the message). The first and last field of the example message in Fig. 1(b) fall in this category: the *Magic* field can be determined as it holds a constant value across messages. The *EXE file* field is also easily identifiable as PE executables have a recognizable structure. We instantiated an initial set of type definitions based on our experience with C&C protocols; this set can be easily extended/customized for specific analysis tasks. Step **C.3** determines *dependent field types*. These fields express properties of other fields or of the message. Our example message contains two: *Message CRC* and *Payload Length* (which in this case expresses the length of the EXE file). Step **C.4** detects *composite field types*, such as lists of content or dependent fields (not present in Fig. 1(b)). Finally, Step **C.5** reconciles the sequence of fields extracted from each individual message into a single *protocol specification* using sequence alignment [7].

III. Protocol Inference

A. Sandboxed execution

The first component of our system investigates whether the malware uses encryption algorithms. It begins by running the malware in a controlled environment consisting of a virtual machine augmented with program tracing (*sandbox* in the following). This stage collects three different program traces:

- A system call trace S, capturing malware-triggered system calls used to send network data (e.g. `send()`). Each call is represented as a set R^S of memory locations and registers used to pass network-bound data to the OS. S therefore consists of a sequence of n such sets $R^S_1 ... R^S_n$.
- An instruction trace T, representing an ordered sequence of m processor instructions executed by the malware. Formally, $T = t_1 ... t_m$, where each instruction t_i is a tuple (i, a, R^T, W^T). i is the instruction opcode, a the instruction address in memory, R^T the set of memory locations/registers read by the instruction, and W^T the set of memory locations/registers written by the instruction.
- A network trace N of all packets generated by the sandbox.

B. Encryption Analysis

Once traces of the malware execution are available, we analyze them to locate instances of encryption use.

1) Candidate Detection: This step executes a *candidate_detection* procedure, which takes as input the instruction trace T and locates a set C of candidate encryption instances. Each candidate in C is a sequence $D \subseteq T$, representing the possible execution of an encryption primitive.

In order to detect encryption in instruction traces several approaches could be used, such as [5], [8], [9]. In our evaluation we use ALIGOT [5] as it is a recent approach with publicly available and well-documented source code. ALIGOT is based on the insight that loops are recurring structures in cipher implementations, therefore searching execution traces for chains of *dynamic loops*. We observed however that the base ALIGOT algorithm does not scale to traces larger than a few tens of thousands of instructions, which prevents a direct application to our case. We concluded that the issue lies in ALIGOT's loop detection, which treats each new instruction as the potential beginning of a new loop, reaching an impractical memory footprint in long sections of straight-line code. After determining that dynamic loops can be seen as *maximal repetitions* [10], we replace ALIGOT loop detection with our implementation of the Kolpakov/Kucherov algorithm [11], which finds all maximal repetitions in a string in linear time. Our optimization lifts the maximum practical trace size from tens of thousands of instructions to a few millions.

2) Candidate Matching: This step executes a *candidate_matching* procedure which receives as input the set of candidate encryption instances C from the previous step, and determines whether each candidate corresponds to the execution of an actual cipher. Formally, *candidate_matching* computes a mapping $C \rightarrow K \cup \{\perp\}$, where K is a set of known ciphers, while $\perp$ represents a failed match. Furthermore, the procedure determines the set of inputs (keys, plaintexts) and outputs (ciphertexts) for each instance. The output of the procedure is a set of matched candidates M, where each element in M is a tuple (D, k, R^M, W^M). $D \subseteq T$ is the sequence of instructions in the candidate encryption instance; $k \in K$ is the encryption cipher being used, R^M is the set of inputs passed to the cipher, and W^M its set of outputs.

It may be possible to implement *candidate_matching* using several possible approaches, such as [5], [12]. We again use the approach proposed by ALIGOT for the same reasons discussed previously. It works by determining all possible input and output parameters to/from the candidate encryption instance, and verifying if any input/output combination matches that of a known cipher. In our implementation we deploy an optimization, by filtering parameters (such as pointers) unlikely to constitute either key material, ciphertext or plaintext.

3) Candidate Filtering: This step consists of a *candidate_filtering* procedure which receives as input the set of matched candidates M emitted by the previous step, and the system call trace S. *candidate_filtering* eliminates elements in M whose output is not used to generate network messages. Recall that each system call in S is described by a set R^S of memory locations and registers used to specify the data to send on the network. A candidate $m = (D, k, R^M, W^M)$ is retained only if $\exists R^S \in S$, s.t. $O = (W^M \cap R^S) \neq \emptyset$ and no other instruction writes the location(s) in O between the execution of m and the execution of the system call R^S. The output is a filtered set of encrypted instances F, with $F \subseteq M$.

We note that there is still potential for false positives from network-bound encrypted data generated by the malware process for non-C&C related purposes. These could be avoided by discarding flows which match known protocols. We leave this additional filtering to future work.

4) Key Type Inference/Decryption: The next goal is to infer general information about how the malware performs encryption. Malware use of ciphers is oftentimes closer to obfuscation than proper encryption [3], employing symmetric ciphers and keys that are either static or provided in the messages. This implies that —once the encryption details have been learned—it is possible to decrypt *any* C&C message within the protocol, not only the ones observed in the sandbox.

We first determine where the outputs of encryption instances in the set F appear in the sandbox-generated network packets from the trace N, by matching the output of encryption instances to packets. We then apply the following two heuristics:

Key type inference: *If the key is constant across all observed message samples, then the C&C protocol is assumed to use the very same key for all possible messages. If the key appears at a constant offset within observed message samples, then the key is assumed to be located at that offset in every message.*

Decryption: *If the observed encryption instances cover full message payloads, the C&C protocol is assumed to fully encrypt each message. If the observed encryption instances only partially overlap with message payloads, it is assumed that the malware encrypts only certain message fields.*

After executing this step, the analyst can provide additional message samples, in order to augment the dataset for protocol analysis. If encryption was detected, all available messages get decrypted. The only exception to this is when the malware only encrypts certain fields. If so, the algorithm has not yet accrued enough information to determine which parts of each message may be encrypted (beyond the ones discovered during trace analysis). In this case, decryption is performed later on a per-field basis, as part of field inference (§ III-C2).

Rule target	Details
String	Detect sequences of printable characters.
IPv4	Match message payload to an IP blacklist.
Timestamp	Detect 4-byte POSIX timestamps (must describe a date within a week of when the message was sent).
PE	Locate Windows binaries with a PE parser.
ZIP	Locate ZIP files. Identify ZIP sections by magic number and merge contiguous sections.
Magic	Detect magic numbers. Looks for a constant prefix appearing in every message.
Padding	Detect message padding (an iterated constant sequence trailing every message).

TABLE I: Content-field detection rules

Formally, the output of this step is a set of decrypted message samples $S = s_1, ..., s_n$. (see Fig. 2(a) for an example).

C. Protocol Analysis

The goal of the next part is to analyze decrypted message samples to infer a *protocol specification* P. For the purpose of our work we define P as a set of n message types $T_1, ..., T_n$. Each T_i is in turn is a sequence of m field types $t_1, ..., t_m$.

1) Message Clustering: This step determines a partition of the sample set S, associating each sample with exactly one message type. We use the insight that most protocols include one or more fields that explicitly specify the message type. We therefore detect likely candidates for message type fields, and cluster messages based on the value of those fields. In order to detect type fields without any prior information about message structure, we use a heuristics due to Bermudez et al. [13], which works by measuring *causality* between candidate fields in pairs of client request/server response messages (the values of such fields are highly likely to be related). We then assign each message to a type based on the value of its type fields.

2) Field inference: After messages have been clustered by type, we proceed to perform *field inference*. The overall goal is to determine a partition of each message sample $s \in S$ in a sequence of fields $f_1, ..., f_n$. Of course, for each field f_i we also want to obtain the corresponding field type t_i.

Our approach is based on a taxonomy of binary protocol fields in four broad categories. The first includes *content fields*, i.e. fields that carry various type of information (IP addresses, executables, etc.) between client and server. The second category includes *dependent fields*, i.e. fields that describe properties of other fields, such as their offset or length. The third category consists of *encrypted fields*, i.e. content and dependent fields that are encrypted. The third category occurs in C&C protocols that perform partial message encryption. We also define *composite fields*, consisting of repeated sequences of basic fields types (e.g. a list of IP addresses).

The core idea of our field inference approach is to detect content fields using a rule-based technique. Once content fields have been located, the algorithm searches for dependent fields. Furthermore, if the protocol performs partial message encryption, the algorithm analyzes the message looking for signs of encrypted content, which is then decrypted to extract additional fields. Finally, the algorithm analyzes the sequence of all discovered fields to identify composite fields.

Content-field inference. The assumption behind this pass is that the structure of many types of content fields is self-evident, i.e. it can be detected by a rule-based approach. For example, consider the example message in Fig. 1(b) after Step C.1. The *EXE file* field detected in Step C.2 has a self-evident structure: a well-defined format recognizable by a PE file parser. We implemented a library of detection rules—detailed in Table I—for various content types, based on our experience with C&C protocols. In order to identify content fields, the detector runs all available rules on message samples.

Rules receive a set of samples and output the detected fields, with no constraints on how they are implemented. We found adding rules to be straightforward: developing the ZIP detector took us two hours with no previous familiarity with the format.

Dependent-field inference. This step breaks each message segment not already covered by a content field in 1-, 2- and 4-byte N-grams. It then checks if the value of each N-gram matches a property of an existing content field or of the message. In the example of Fig. 1(b), Step C.3, two dependent fields are detected: *CRC* and *LENGTH*. Table II lists the properties each N-gram is checked against.

Encrypted-field inference. A special case of fields are encrypted fields in partially encrypted protocols. In order to identify such fields within the rest of a message, we devise a technique based on the observation that the ciphertext of an encrypted field appears as a random bitstring, while the corresponding plaintext exhibits some type of structure.

We use a randomness test by Goubault-Larreq and Olivain [14]. Briefly, the test works by estimating the Shannon entropy H_N^{MLE} of a sequence of N bytes using the maximum-likelihood estimator, and comparing it with the expected value H_N of the N-truncated entropy of a uniformly random sequence of the same length. If the distance is less than a predefined threshold, the sequence of bytes under examination is judged random (see [14] for details). Consider a procedure *randomness_test* that performs such a test, and a message M. For each possible byte offset $o = 1...n$ in the message and for each possible field length l, our detector computes $randomness\_test(M[o : o + l])$ and $randomness\_test(D(M[o : o + l]))$ (where D is the decryption function). Whenever the original byte sequence is random—according to the test—but the decrypted one is not, the decrypted value is passed to the content- and dependent-field detectors.

Composite-field inference. Protocol specifications may include *set semantics* [15], i.e. repeated sequences of one or more field types; different messages of the same type may carry sequences of different length. Our approach therefore incorporates a detector—based on the Kolpakov/Kucherov algorithm (§ III-B1)—to find repetitions in the sequence of discovered fields. Sequences of different lengths but carrying the same type(s) are mapped to the same composite field type.

Overall, the field inference steps output a set of messages, each labeled with a set of discovered fields (e.g. see Fig. 2(b)).

5) Specification Abstraction: The final step processes all messages in each type cluster, isolating the commonalities

Rule target	Details
Length	Detect fields whose value F satisfies $F = aL - b$ for some a, b (L is a field or message length).
Offset	Detect fields whose value F satisfies $F = aO$ for some a (O is the offset of a known content field).
CRC	Detect fields whose value matches $H(M)$ (M is the message content and H a known hash function).

TABLE II: Dependent-field detection rules

between sequences of fields in different messages, and abstracting a message-type specification T (recall that such specification consists of a sequence of field types $t_1, ..., t_m$). This step is necessary to reconcile inconsistencies in the fields detected in different messages of the same type, which arise for two reasons. First, message clusters determined in Step C.1 may be imprecise, or a protocol may use different message formats within the same declared message type. Second, rule-based heuristics may not detect all occurrences of a field.

In order to merge multiple field sequences in a single specification, we use the Needleman-Wunsch algorithm [7]—a dynamic programming technique to find the optimal alignment of similar sequences, often used for message comparison and clustering (e.g. [15], [16]). A significant issue is that the coverage of message samples may not be complete, as each message may include one or more sections in which the field inference step was not able to detect any field. When performing alignment, we allow sequences to include *blanks* representing one or more unknown field. We treat blanks as wildcards, allowing them to align with any concrete field type.

For each message type, we choose the message sample for which the richest set of fields has been inferred, and we generate a specification based on that. Then, we iteratively align the field sequence in the specification with the field sequences in all other messages of the same type. Every time the two sequences align, we "fuse" them in a single sequence specification (if a blank aligns with a concrete field type, the merged specifications only includes the latter). If a cluster contains messages that do not align with the specification, we split it and repeat specification abstraction on both the newly created clusters, recursively splitting them again if necessary.

The final output is the protocol specification (e.g. Fig. 2(c)).

D. Signature Generation

One relevant use for malware protocol specifications is the generation of network signatures. We therefore also propose a methodology to generate deterministic signatures, which verify an ordered set of *tests*, each of which checks the presence of an expected field. We declare a match if all the (ordered) tests in a signature are satisfied. The tests are rules that check particular field characteristics in a given group of contiguous bytes, defined by an offset and length. We develop tests for all the content-field detectors in Table I. For example, the *MATCH_STRING* test will check for the existence of only printable characters in the group. In addition, we developed tests to check for the dependent fields in Table II. For instance, *MATCH_MSG_LENGTH* will check that the message length is equal to the value of the input byte group.

IV. Evaluation

We evaluate our approach in depth showing that it can: (i) correctly identify encryption uses, (ii) effectively infer the structure of C&C messages, (iii) generate correct protocol specifications, and (iv) lead to effective network signatures.

A. Malware Families under Examination

For our evaluation, we selected three malware families based on: (i) the level of understanding of the C&C protocol by the research community, so to have a well-defined ground truth; and (ii) the popularity of the malware in recent times. For each family we collected and run one binary sample. The families are: **Sality [17]**, a polymorphic file infector and downloader with complex features, such as P2P support (we focus on the Sality.AE variant). **Ramnit [18]**, a malware platform with various capabilities, provided through a flexible plugin architecture. Ramnit's botnet is estimated to have included 3M+ infected machines at his peak [19]. **ZeroAccess [6]**, a trojan that can act as a backdoor and download additional malware. It uses P2P to participate in botnets extending to tens of thousands of infected machines [20].

B. Infrastructure

Our sandbox implementation (Step A in Fig 1(a)) is based on the Cuckoo sandbox [21], extended with a custom analysis package based on PIN [22] to generate instruction traces[1].

The sandbox output (instruction trace, syscall trace and network trace) is passed to our encryption analysis tool (Steps B.1-B.4). The candidate detection and candidate matching steps in our algorithm are based on the publicly available ALIGOT Python implementation [23], modified to introduce our optimizations. Separately, the tool analyzes the sandbox-generated network trace to extract malware messages, and matches extracted messages with detected encryption uses; if possible each message is then decrypted (Step B.4). This step returns a list of message samples annotated with encryption information (Fig. 2(a)).

A second application implements message clustering, field inference, and abstraction (Steps C.1-C.5). The output consists of (i) a list of messages, each augmented with a list of detected fields (Fig. 2(b)), and (ii) specifications for each detected message type (Fig. 2(c)). Overall, our toolchain consists of 17500 lines of Python code and 1900 lines of C++.

C. Additional Datasets

In order to augment the diversity of our C&C dataset beyond the messages obtained from the sandbox, we integrated additional message datasets, detailed in Table III (user-generated traces were anonymized). To extract C&C messages from the network traces, we used a combination of Snort [24], custom scripts and manual inspection. We emphasize that message extraction was a non-trivial effort, spawning several weeks of analyst's time. Benign traffic samples used for our signature matching evaluation (§ IV-H) are also detailed in Table III.

[1] We used industry-standard tools for VM tuning and hardening (e.g. pafish) to circumvent malware anti-VM techniques.

```
ORIGIN: Ramnit
L4 PROTOCOL: TCP
SOURCE IP: 192.168.1.1
SOURCE PORT: 49197
DESTINATION IP: 192.168.1.2
DESTINATION PORT: 443
DIRECTION: Client → Server
TIMESTAMP: 1435260024.859
PAYLOAD: 00FF4B000000E20020000(...)
ENCRYPTION INFORMATION:
  USE #1: cipher="RC4" key="626C61636B"\
          length="32" offset="49"\
          plaintext="353437353563(...)"
  USE #2: cipher="RC4" key="626C61636B"\
          length="32" offset="12"\
          plaintext="306263356665(...)"
```

(a) Output of decryption (step B.4)

```
ORIGIN: Ramnit
PAYLOAD: 00FF4B000000E20020000(...)
FIELDS INFORMATION:
  FIELD #1: type="MAGIC" position="0-1"
  FIELD #2: type="MSG_LENGTH" position="2-5"
  FIELD #3: type="MSG_TYPE" position="6"
  FIELD #4: type="FIELD_LENGTH"\
            position="8-11" refers-to="5"
  FIELD #5: type="ENCRYPTED_RANGE"\
            position="12-43"
     FIELD #5.1: type="STRING"\
                 position="12-43"
  FIELD #6: type="FIELD_LENGTH"\
            Position ="45-48" refers-to="7"
  FIELD #7: type="ENCRYPTED_RANGE"\
            position="49-80"
     FIELD #7.1: type="STRING"\
                 position="49-80"
MESSAGE TYPE: E2
```

(b) Output of field inference (C.1-C.4)

```
MESSAGE_TYPE: "E2-CS-0"
FIELD #1: type="MAGIC" length="2" content="00FF"
FIELD #2: type="MSG_LENGTH" length="4"\
          content="4B000000"
FIELD #3: type="MSG_TYPE" length="1" content="E2"
FIELD #4: type="BLANK" length="1" content="00"
FIELD #5: type="FIELD_LENGTH" refers-to="6"\
          length="4" content="20000000"
FIELD #6: type="ENCRYPTED_RANGE"length="32"\
          content_type="variable"
   FIELD #6.1: type="STRING" length="32"\
               content_type="variable"
FIELD #7: type="BLANK" length="1" content="00"
FIELD #8: type="FIELD_LENGTH" refers-to="9"\
          length="4" content="20000000"
FIELD #9: type="ENCRYPTED_RANGE" length="32"\
          content_type="variable"
   FIELD #9.1: type="STRING" length="32"\
               content_type="variable"
```

(c) Output of specification abstraction (C.5)

Fig. 2: Examples of output of various algorithmic steps for the Ramnit C&C protocol

Traces used to extract C&C samples		
Trace	**Size**	**Type**
ISP	635GB (330M flows)	User-generated traffic from large European ISP
MTA	386MB (17K flows)	Malware-generated traffic from controlled environment [25]
Traces containing benign traffic		
DARPA	3.4GB (858K flows)	DARPA corpus [26] (week #2)
M57	4.6GB (93K flows)	Corpus for forensics training [27]

TABLE III: External traces used in the evaluation

D. Encryption Analysis

In this section, we evaluate how effective is our encryption analysis algorithm in: (i) detecting the use of encryption, and (ii) inferring details of the encryption used in C&C protocols.

In order to evaluate the detection part, we define an *encryption instance* as a contiguous block of message bytes encrypted with the same key. We ran the malware binaries, collected instruction traces and fed them to our analysis infrastructure (Steps B.1-B.3). We then compared the number of detected instances to the true number determined via manual analysis. Table IV, col. 2 shows that our approach detects all encryption instances for all malware families. As repeated experiments led to the same outcome, we present the result of a sample run.

We then fed the detected encrypted instances and the sandbox-generated network traces to our encryption inference algorithm (Step B.4). In all cases, the tool mapped instances to C&C messages, and correctly inferred encryption details of interest (see Table IV col. 3-5). Overall, results in this section suggest that our approach is effective in detecting and assessing uses of encryption in C&C protocol messages.

E. Message Clustering/Inference

This part of our evaluation investigates the effectiveness of our approach in: (i) clustering C&C messages by type, and (ii) detecting various field types within messages. In this step, we integrated the set of malware messages obtained from the sandbox with samples from the traces in Table III. The aggregated dataset consists of 20615 Sality messages, 1237 Ramnit messages and 49378 ZeroAccess messages.

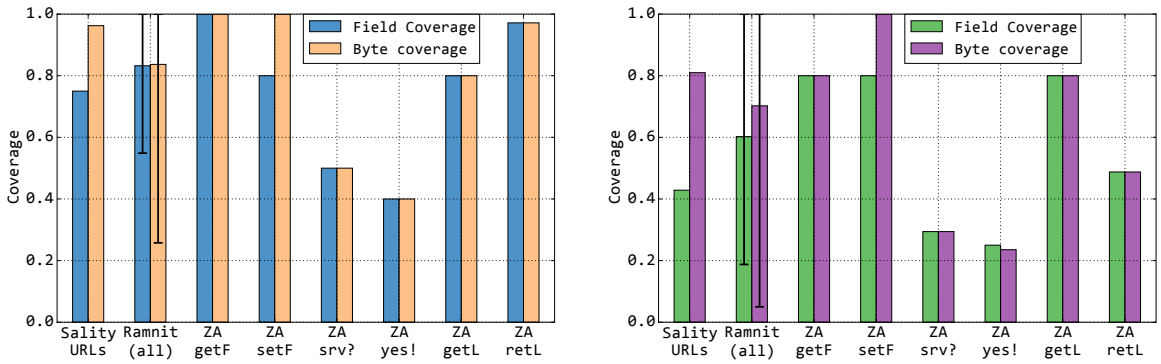
Clustering: This step detects message-type fields and partitions the message set based on these fields. For Sality, our trace only includes client-to-server messages, which prevents the message-type heuristic (§ III-C1) from being applied. All messages are therefore mapped to a single type, which is correct, as all messages are of the *pack exchange* variety (used by peers to spread IPs of other bots). In the case of Ramnit and ZeroAccess full conversations are available. In both cases the heuristics isolates the relevant message-type bytes, identifying *23 of 29 true message types for Ramnit, and 4 out of 4 true message types for ZeroAccess*. The discrepancy in Ramnit message types is due to messages with the same declared type but different structure, perhaps reflecting protocol evolution.

Field inference: This step analyzes messages in each type cluster. Rule-based detectors are used to identify various types of field (ref. Tables I and II). We define two metrics to assess the effectiveness of field inference: for a set of samples of a given type, *field coverage* is the fraction of correctly-inferred fields, while *byte coverage* is the fraction of bytes within correctly inferred fields. We consider a field to be correctly inferred if its type/position align with that of a true field.

We note that the C&C protocols evaluated include several field types not supported by our approach. This is because we only support fields that map to generic data types (e.g. string, IPv4, CRC) and ignore malware-specific data types (e.g. ZeroAccess messages carry binary values expressing the age in seconds of peer IP addresses).

Fig. 3(a) presents field and byte coverage of the set of supported fields for each message type (i.e. the baseline only includes fields whose type is supported by our detectors). For Ramnit we only present aggregate results—average/min/max—as this malware defines 29 distinct types. Type names are taken from protocol specifications.

In general, our approach detects the majority of supported fields/bytes. The limited coverage of certain ZeroAccess message types is due to timestamps which predate their respective messages by more than one week, causing the timestamp detector to ignore them. Sality's undetected content consists of message-type fields, that cannot be identified since only client-side messages are available. Note that that most message types contain < 10 fields, therefore even a limited number of undetected fields can greatly affect the result. Overall we achieve 97% supported-field coverage (amounting to 97% of



(a) Coverage of supported field types

(b) Overall coverage

Fig. 3: Coverage of supported field types and of full message content for each message type

Malware	Instances found/present	Key material	Cipher	Encryption type
Sality	64/64	Carried in message	RC4	full message
Ramnit	32/32	Constant	RC4	partial(some fields only)
ZeroAccess	120/120	Constant	RC4	full message

TABLE IV: Details of the encryption used by each malware

bytes in those fields), with null or negligible false positive rate.

Fig. 3(b) presents overall coverage of all message content, including fields for which no detector is available. The large majority of undetected fields/bytes is taken up by ZeroAccess' peer-age fields described above. Such semantics are malware-specific and cannot be captured by generic rules.

In summary, results show that our approach can identify a significant fraction of the structures carried by C&C messages.

F. Specification Abstraction

In order to evaluate the correctness of auto-generated specifications we compare them to manually-generated specifications based on domain knowledge and analyses of the C&C protocols ([17], [28], [29]). Results are presented in Table V.

Column #3 in the table shows the number of inferred types in each case. In most cases, there is a 1:1 mapping between inferred and actual types; the exception is the ZeroAccess *retL* message type, for which our algorithm abstracts 2 different message sub-types. Messages of this type carry sequences of *(Peer IP address, Peer age)* pairs. The composite-field heuristics detect a sequence of fields only if it includes three or more elements; hence, sequences of length 2 are mapped to a dedicated field type.

Column #4 lists the number of supported field types in the specification of each message type. Columns #5 describes how many supported fields were correctly recognized for each message type. Column #6 details false positives: fields that were inferred but are not present in the protocol specification.

Overall, our approach produces informative specifications, consistent with the structure of the protocol under analysis.

G. Performance

Table VI details the performance of all steps of our algorithm when running on a Linux Server with Intel Xeon 2.50 GHz processors and 64 GB of RAM. We do not present sandbox execution times as they depend on the malware under analysis (e.g. our Sality binary waits 10 minutes before generating traffic). We emphasize that although our current implementation is single-threaded, each step presents abundant opportunities for parallelism. For example, analysis of distinct messages could be largely performed in parallel.

H. Signature Generation and Matching

For each of the C&C message sets, we generated (using the methodology described in § III-D) and tested signatures using *5-fold cross-validation* (4 partitions for training and 1 for testing in every iteration). The signature generation and matching logic is implemented as a set of PERL scripts, totaling 700 lines of code. This code receives as input (i) protocols specifications generated by our toolchain, and (ii) a set of messages extracted from network traffic. Each message is then either matched to a malware protocol specification, or discarded as benign if no match is found.

Table VII shows, for the three malware families, the average percentage of messages of a given family that matches the signatures. We also show a column for *Others*, which include all the traffic from the malware-free traces DARPA and M57. These traces consist of a mix of popular protocols (DNS, HTTP) and unknown UDP and TCP traffic. Overall, our results show no false positives and very low false negatives.

V. Discussion

Per-session encryption and TLS: In order to decrypt message samples from external sources, our approach requires that encryption key and/or the decryption approach inferred from sandbox-originated traces, are valid for *any* protocol message. It is in principle possible to design protocols that generate per-message keys which are non-trivial to infer, or use asymmetric encryption. One case of particular interest within this category is that of protocols using TLS. First, we note that as long as the cipher can be detected, our approach can be extended to extract the plaintext for messages generated/received by the sandbox. However, since each connection dynamically negotiates unique keys, it is no longer possible to infer a botnet-wide encryption key. Decryption may still be achieved via protocol-specific approaches; e.g. decrypt TLS connections via TLS proxying.

Malware	Msg types (ground truth)	#Msg types (inferred)	#Supported fields (ground truth)	#Supported fields (correctly inferred)	#Supported fields (false positives)
Sality	URLs	1	4	3	0
Ramnit	29 types	28	168	124	6
ZeroAccess	getF	1	5	4	0
	setF	1	5	4	0
	srv?	1	10	5	0
	yes!	1	10	4	0
	getL	1	5	4	0
	retL	2	266	262	2

TABLE V: Breakdown of inferred specifications

In our work, we observed that malware commonly tends to use simple forms of symmetric encryption, and to reuse keys. For example, our C&C dataset for the popular Ramnit malware shows that the same key was used in 2012 and 2014.

Text-based protocols: Our approach focuses on binary-only protocols due to their popularity, however it is possible for malware to use text-based protocols. Although supporting them is outside the scope of this work, we note that the encryption analysis component can still be applied.

Session semantics: Our current analysis does not investigate how multiple messages are organized into sessions. Although inferring the protocol state machine (such as in [30]) is outside the scope of our work, our approach could be easily extended to infer various types of session-related information. In particular, our approach partitions each malware connection into an ordered sequence of client and server messages, enabling inference of the conversation structure (e.g. *Client messages of type A are always followed by server messages of type B*). It would be similarly possible to detect *session keys*—fields that are common across requests and responses—e.g. using the heuristics from [13]. We leave this as future work.

VI. Related Work

Protocol reverse engineering: Techniques proposed for protocol reverse-engineering can be divided into three groups, depending on their input data. The first one generates protocol specifications from network traces. For instance, ScriptGen [31] generates a state machine from the network traffic observed, in order to produce scripts that approximate responses to different protocol requests, while [15] automatically infers message formats using a few very generic pre-defined field semantics. None of these approaches targets binary protocols; furthermore, we infer detailed specifications which include rich field types, and we deal with encryption while previous works do not.

The second group of works in this space includes techniques that analyze execution traces captured as a program is communicating over the network. [32] presents a technique to generate specifications by instrumenting a program during message processing. The approach focuses on server-side code, which in the malware world is notoriously difficult to access. Dispatcher [33] does automatic reverse-engineering of C&C protocols by performing dynamic analysis on the malware binary. By tainting the incoming data from the network, the authors can derive detailed semantics of fields after identifying the prototype of the function in which the data will be used. For outgoing data, the authors employ a series of heuristics to determine the position, size and semantics of fields. The authors also consider protocols that use encryption, by leveraging existing techniques [34], where plain-text data is extracted from the input (output) buffers to encryption (decryption) functions. We differ from Dispatcher in two important ways: (i) we can learn and generalize encryption details, enabling analysis of samples not generated in the sandbox, and signature generation and online traffic decryption; (ii) we only require lightweight, passive binary analysis which can be avoided if a malware from the same family has been analyzed in the past.

The third group of works follows a hybrid approach. Prospex [35] analyzes both network traffic and execution traces to infer the malware's protocol state machine. However, [35] does not handle encrypted traffic.

Botnet C&C traffic detection: Several approaches in literature focus on generating unique patterns or signatures to identify C&C protocol communications. ProVex [3] is probably closest to our work. It heuristically attempts to decrypt packets using known encryption algorithms and extracts a statistical profile of the byte distribution in the payload. Differently from our approach, it requires previous knowledge of encryption keys. In addition, its signatures are probabilistic and thus potentially prone to higher false positives, while we can build very detailed signatures based on field semantics. Botzilla [36] works similarly but does not perform decryption, assuming instead that recurring textual features can be found even within encrypted messages. CoCoSpot [37] also targets C&C protocols and deals with encryption traffic by using features that are orthogonal to payload obfuscation, such as message lengths. FIRMA [38] generates malware signatures that primarily target HTTP-based C&C communication. Similarly, Perdisci et al. [39] uses URL-based features to cluster malware-derived HTTP samples and generate effective signatures (as opposed to our approach, which focuses on binary messages). Both [38], [39] rely on deriving sets of constant tokens via string analysis, which is not possible in the presence of encryption.

C&C Server Fingerprinting: These kinds of works try to identify C&C servers by analyzing their response to carefully constructed probes. Two recent works in this area are [40], [41]. These works aim at generating probes that elicit a meaningful answer from the server, while we aim at inferring full specifications for message formats to enable detection.

Malware	#Instructions in program trace	Trace analysis	#Msg samples in dataset	Avg. #fields per message	Enc. inference/ Msg decryption	Clustering	Field inference	Specification abstraction
Sality	67M	103min	20615	6	2s	2s	8s	8s
Ramnit	13M	16min	1237	7	<1s	<1s	5min	<1s
ZeroAccess	19M	20min	49378	131	3s	44s	73min	36s

TABLE VI: Dataset sizes and execution times of algorithmic steps

	Average % of Messages Detected per Signature			
Malware	Sality	Ramnit	ZeroAccess	Others
Sality	100	0	0	0
Ramnit	0	96.4	0	0
ZeroAccess	0	0	99.9	0

TABLE VII: Traffic classification results

VII. Conclusion

In this paper we present a solution to infer the format of malware binary C&C protocols. Understanding such protocols is crucial to gain insight into how malware works, and to build effective network-based malware detectors. Our inference algorithm produces detailed protocol specifications, including rich content types—significantly alleviating the cumbersome and error-prone task of understanding malware communications manually. Our approach works in the presence of encryption, using program analysis to recover encryption keys. Evaluation results show that our approach is effective in decrypting malware samples and infers rich specifications, which generate effective network signatures for C&C protocols.

References

[1] "Internet Security Threat Report, Volume 20," Symantec, Tech. Rep., Apr. 2015.

[2] "Safeguarding The Internet: Level 3 Botnet Research Report," Level 3 Communications, Tech. Rep., Jun. 2015.

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

[4] J. C. Mitchell, *Foundations of Programming Languages*. Cambridge, MA, USA: MIT Press, 1996.

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

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

[7] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, Mar. 1970.

[8] R. Zhao, D. Gu, J. Li, and R. Yu, "Detection and analysis of cryptographic data inside software," in *ISC*, 2011.

[9] F. Gröbert, C. Willems, and T. Holz, "Automated Identification of Cryptographic Primitives in Binary Programs." in *RAID*, 2011.

[10] M. Lothaire, *Applied combinatorics on words*. Cambridge University Press, 2005.

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

[12] P. Lestringant, F. Guihéry, and P.-A. Fouque, "Automated identification of cryptographic primitives in binary code with data flow graph isomorphism," in *ASIA CCS*, 2015.

[13] I. Bermudez, A. Tongaonkar, M. Iliofotou, M. Mellia, and M. M. Munafo, "Automatic Protocol Field Inference for Deeper Protocol Understanding," in *IFIP*, 2015.

[14] J. Goubault-Larrecq and J. Olivain, "Detecting Subverted Cryptographic Protocols by Entropy Checking," ENS-Cachan, Tech. Rep. LSV-06-13, 2006.

[15] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in *USENIX*, 2007.

[16] M. A. Beddoe, "Network protocol analysis using bioinformatics algorithms," Tech. Rep., 2004.

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

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

[19] "Europol cracks down on botnet | Ars Technica." [Online]. Available: http://arstechnica.com/tech-policy/2015/02/europol-cracks-down-on-botnet-infecting-3-2-million-computers/

[20] "ZeroAccess botnet resumes click-fraud activity after six-month break | Dell SecureWorks Blog." [Online]. Available: http://www.secureworks.com/resources/blog/zeroaccess-botnet-resumes-click-fraud-activity-after-six-month-break/

[21] "Cuckoo Sandbox." [Online]. Available: http://cuckoosandbox.org/

[22] 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.

[23] "Aligot source code," May 2016. [Online]. Available: https://code.google.com/archive/p/aligot/

[24] "Snort IDS," May 2016. [Online]. Available: http://www.snort.org/

[25] "Malware-Traffic-Analysis.net." [Online]. Available: http://malware-traffic-analysis.net/about.html

[26] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das, "Analysis and results of the 1999 DARPA off-line intrusion detection evaluation," in *RAID*, 2000.

[27] "M57-Patents Scenario," May 2016. [Online]. Available: http://digitalcorpora.org/corpora/scenarios/m57-patents-scenario

[28] C. Chen, "Virus Bulletin: Ramnit bot," Nov. 2012. [Online]. Available: https://www.virusbtn.com/virusbulletin/archive/2012/11/vb201211-Ramnit.dkb

[29] K. McNamee, "Malware analysis report - botnet: ZeroAccess/Sirefef," Kindsight, Tech. Rep., Jul. 2014.

[30] C. Y. Cho, E. C. R. Shin, D. Song, and others, "Inference and analysis of formal models of botnet command and control protocols," in *CCS*, 2010.

[31] C. Leita, K. Mermoud, and M. Dacier, "ScriptGen: an automated script generation tool for Honeyd," in *ACSAC*, 2005.

[32] G. Wondracek, P. M. Comparetti, C. Kruegel, and E. Kirda, "Automatic Network Protocol Analysis," in *NDSS*, 2008.

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

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

[35] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol Specification Extraction," in *IEEE S&P*, 2009.

[36] K. Rieck, G. Schwenk, T. Limmer, T. Holz, and P. Laskov, "Botzilla: detecting the phoning home of malicious software," in *SAC*, 2010.

[37] 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.

[38] M. Z. Rafique and J. Caballero, "Firma: Malware clustering and network signature generation with mixed network behaviors," in *RAID*, 2013.

[39] R. Perdisci, W. Lee, and N. Feamster, "Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces." in *NSDI*, 2010.

[40] A. Nappa, Z. Xu, J. Caballero, and G. Gu, "CyberProbe: Towards Internet-Scale Active Detection of Malicious Servers," in *NDSS*, 2014.

[41] Z. Xu, A. Nappa, R. Baykov, G. Yang, J. Caballero, and G. Gu, "Autoprobe: Towards automatic active malicious server probing using dynamic binary analysis," in *CCS*, 2014.