Reconstructing C2 Servers for Remote Access Trojans with Symbolic Execution

Luca Borzacchiello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu

Sapienza University of Rome
{borzacchiello,coppa,delia,demetres}@diag.uniroma1.it

Abstract. The analysis of a malicious piece of software that involves a remote counterpart that instructs it can be troublesome for security professionals, as they may have to unravel the communication protocol in use to figure out what actions can be carried out on the victim's machine. The possibility to recur to dynamic analysis hinges on the availability of an active remote counterpart, a requirement that may be difficult to meet in several scenarios. In this paper we explore how symbolic execution techniques can be used to synthesize a command-and-control server for a remote access trojan, enabling in-vivo analysis by malware analysts. We evaluate our ideas against two real-world malware instances.

Keywords: Malware analysis, symbolic execution, protocol reversing.

1 Introduction

Remote Access Trojan (RAT) is a term used to identify a cyber menace that can steal information and carry out malicious behaviors on a victim machine at the command of a remote counterpart. RATs provide attackers with capabilities like file upload, key logging, and remote code execution, communicating with a counterpart commonly dubbed C2 (Command & Control) server. Communication protocols are specific to RAT families and can become richer as families evolve over time: although packets are carried out using standard means like HTTP or IRC, their format is proprietary and the contents possibly encrypted.

Unlike other malware categories, analyzing a RAT typically requires active ongoing communications for the results of the analysis to be rich. This is the case not only with an initial assessment in a sandbox, but also with an in-depth analysis by malware analysts on its code. Unfortunately this may not be possible for a variety of reasons. For instance, a company that discovers and promptly contains an infection [13] may not allow communications for the sake of analysis, especially where there is a suspicion of a targeted attack—as such communications would reveal that the target has been reached. A server counterpart may also decline connection attempts when the analysis takes place from an unexpected network origin or time frame. Another common scenario is when the server disappears from the network—either at the attacker's will or because of an intervention by the authorities—but the analysis of the RAT could still be valuable (e.g., to unveil its infection spreading mechanisms).

Automatic analysis systems may be of little use in the absence of an active counterpart, leaving manual dissection as the only avenue to analysis. RATs can accept dozens distinct commands as part of the communications with the C2 server: while commands are usually not hard to analyze when taken individually, their number and interplay can make the analysis quite complex and time consuming. Authentication schemes can make the work of human analysts even more tedious as they have to mimic them when carrying out the analysis. Finally, even in the presence of an active counterpart the server may not be sending all the supported commands within a reasonable timeframe, leaving to the analyst the task of unveiling the characteristics of the remaining ones.

Protocol reverse engineering techniques may reveal details of the communication scheme, producing automata and grammars that capture facts inferred by analyzing packets. Their applicability to RAT analysis may however be limited by constraints, e.g., on having to communicate with the server for validating messages, or on having access to its binary. Although the output of such techniques can bring valuable insights to analysts, they could not be used readily in the setting where malware dissection normally takes place, which would instead benefit from having a synthesized counterpart to interact with in order to reveal indicators and features of the sample other than its communication protocol.

Previous research has explored static analysis techniques to ease RAT dissection: in particular, [2] proposes symbolic execution to reveal and analyze the commands supported by a RAT without requiring the presence of the server counterpart. The output is a collection of execution traces enriched with symbolic constraints on the data buffers exchanged throughout communications. Unfortunately, such traces typically encompass at most one command due to scalability issues, and may come in numbers often much greater than the supported commands due to the nature of the symbolic exploration: for instance, multiple paths can be generated for the same command when its handler contains loops or when memory is dereferenced using a symbolic pointer [3, 10]. Another problem is that such reports cannot be merged to form longer paths descriptive of communication patterns with the C2 server, as protocol rules may shape subsequent packets differently than in a naive concatenation of recorded communications (consider for instance the use of sequence numbers).

Contributions. We propose a technique to reconstruct the C2 server counterpart for a RAT by having access only to the sample meant to run on the victim's machine. Our goal is to synthesize a program so that the analyst can make the client connect to it, enabling in-vivo analysis of the RAT. To this end:

- we produce execution traces for a sample using symbolic execution, exploring the full software stack to avoid writing API models required by [2];
- we assemble traces into a compact automata representation inspired by previous protocol reverse engineering research;
- we augment the automaton with speculative edges not seen in recorded traces to build paths that span multiple commands in a C2 communication session, and validate them with symbolic execution to generate server instances.

We implement the technique in the context of the S2E platform [6], and perform a preliminary assessment against two well-known RAT samples.

2 Approach

In the present section we provide the reader with an overview of our approach. We then describe in detail every step of the proposed technique, using a simple C2 communication protocol implemented in an artificial RAT as running example to highlights challenges and key points in our strategy. Finally, we discuss open problems and limitations of our solution.

2.1 Overview

Figure 1 visually summarizes the main steps behind our approach:

1. *Trace Generation.* Given a RAT sample, we use symbolic execution [3] as in [2] to reveal different executions paths, trying to maximize code coverage during the exploration. For each path we record traces of taken branches and relevant APIs that get executed. The presence of input-dependent loops and the uncertainty on other portions of the input may lead to a high number of alternative states in the symbolic executor, but we expect a significant fraction of the recorded traces to reveal at least one command of the RAT (i.e., the associated paths are deep enough to perform authentication and carry out one of the supported functionalities). Traces where no interesting APIs have been executed are discarded.
2. *Trace Analysis.* To reason over the possibly large number of reports generated during the previous step we build an Augmented Prefix Tree Acceptor (APTA) [5], a tree-shaped DFA used in previous research on protocol reverse engineering [9]. This representation allows us to compactly represent traces into a single data structure, capturing which APIs and branches have been observed along different paths.
3. *Speculation.* While an APTA can capture information about distinct (symbolically executed) paths, our goal is to generate a small number of paths—ideally one—that can cover all the commands. We use information from past executions to speculate over richer paths that could be possible in the RAT but were not observed possibly due to the limited scalability of the symbolic analysis. To this end, we add speculative edges to the APTA—turning it into a graph—and compute path(s) that can cover all its nodes, describing an execution where multiple commands are taken in by the RAT.
4. *Validation.* We validate the feasibility of a speculatively generated path with symbolic execution, using knowledge on the branches from initially recorded traces to drive the exploration. We rely on the symbolic engine to fill the blanks for the actions that take place between two commands linked by a speculative edge, as the latter is not backed by a trace. When the symbolic exploration succeeds, we record a sequence of API calls and results that in a native execution would lead the client to follow the path. When we cannot produce a valid sequence, we go back and try a different speculation.

5. *C2 Server Generation.* For each validated path spanning multiple commands, we synthesize a C2 server counterpart for executing the client natively. At each step the server executes APIs that symmetrically interact with the ones invoked by the client (e.g., they send data when a client is expecting to receive some), using inputs that were instantiated in the symbolic exploration.

2.2 Steps

To illustrate the different steps of our approach we will use the artificial RAT instance of Figure 2 as running example: as we mentioned, its code is meant to capture common traits of several real-world RATs. The client instantiates a connection to the C2 server and waits from the attacker for the commands to be executed on the victim's machine. Its dispatcher takes the form of an infinite loop for processing incoming packets: our RAT supports only two commands, namely the first allows the attacker to execute a program using the `WinExec` Windows API, while the second can remove a file using `DeleteFile`. A simple authentication scheme is implemented at branches B2 and B3 in the code, while sequence numbers are used for packet validation at branch B4.

Trace Generation. Similarly as in [2] we use symbolic execution techniques to reveal interesting control-flow paths that the execution may follow within a RAT. A symbolic engine supplies symbolic expressions in place of concrete inputs to a program, where an input may represent data coming from the network or other interactions with the OS. When symbolically executing instructions in the program, the engine maintains a collection of constraints on the program state, forking the execution when a branch involving symbolic expressions is met and both outcomes are feasible. An SMT solver is used to evaluate symbolic expressions as well as to obtain valid concrete assignments for them.

Symbolic execution can be very valuable in our setting, as the analysis can be carried out by modeling the communications with the C2 server using symbolic buffers in network-related APIs. Unfortunately though symbolic techniques incur scalability issues (especially the path explosion problem [3, 2]) that make it hard to analyze long program paths. Fully automatic approaches to the exploration of a RAT are unlikely to reveal the commands without heuristics tailored to the structure of the sample under analysis; also, the presence of code for its initial activities (e.g., environmental checks, achieving persistence) may prevent the exploration from reaching the command dispatching loop at all. For this reason we ask the analyst to provide hints in the form of the location of the dispatcher loop and relevant details for reaching it (such as the thread on which the analysis should focus or branches that were possibly forced in the debugger).

Once the symbolic exploration reaches the loop for the first time, we adopt an iterative deepening search strategy as in [2] to combine the benefits of BFS and DFS in this specific setting: each alternative path originating from the uncertainty on received inputs is explored for a given number of instructions before the exploration switches to another path in the queue. Optionally the analyst

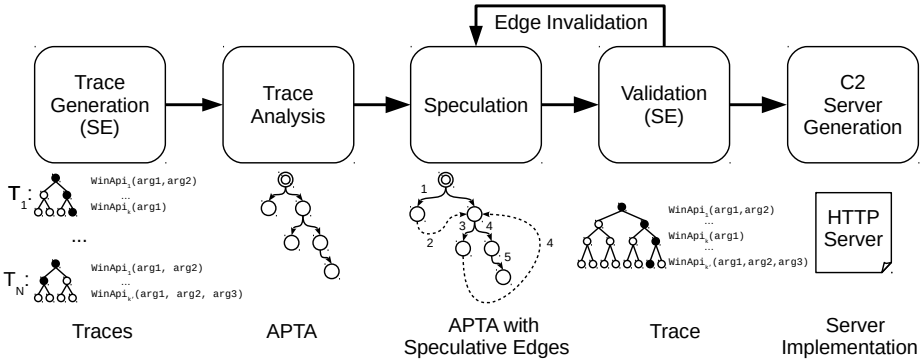

Fig. 1: Overview of the steps for the proposed approach.

```
unsigned char seqNum = 0;
unsigned char data[N];
int socket = connectWithServer(); // socket + connect
while (1) { // B1
    clearBuffer(data, N);
    read(socket, data, N);
    if (seqNum == 0) { // B2
        if (strncmp("MAGIC_STR", data, N) == 0) // B3
            seqNum += 1; // authentication success
    } else if (seqNum == data[0]) { // B4
        switch (data[1]) {
            case 1: // B5
                WinExec(data + 2, 0); break;
            case 2: // B6
                DeleteFile(data + 2); break;
        }
        seqNum += 1;
    }
}
```

Fig. 2: Artificial RAT instance. For the sake of presentation, conditional branches and `case` statements in the C code are annotated with labels $\mathtt{B}_K$.

can provide further hints on the location of the handler for individual commands, which may be simple to spot in a debugger unless the code is heavily obfuscated. For each path we record the sequence of taken branches and Windows API calls for communications or anyhow relevant in malicious code analysis (we will refer to such APIs as *interesting*). Branches are split in groups across invocations of APIs that produce new symbolic inputs. The exploration is terminated upon exhaustion of the memory and time budget for the analysis.

When symbolically executing our artificial RAT using the search strategy described above, and assuming that we exhaust the time budget after completing two iterations of the dispatcher loop (branch B1) along each possible path, our approach generates the following traces:

– *Trace 1*
  (a) API: `socket(...)`, `connect(..., SERVER_ADDR, ...)`, `read( ..., D1, 64)`, `read(..., D2, 64)`, `WinExec(D3, 0)`

(b) buffers: `*D1:= "MAGIC_STR", *D2:= "11Program.exe", *D3:= "Program.exe"`
(c) taken branches: `[B1], [B2, B3], [B4, B5]`

– *Trace 2*
(a) API: `socket(...), connect(..., SERVER_ADDR, ...), read( ..., D1, 64), read(..., D2, 64), DeleteFile(D3)`
(b) buffers: `*D1 = "MAGIC_STR", *D2 = "12File.doc", *D3 = "File.doc"`
(c) taken branches: `[B1], [B2, B3], [B4, B6]`

– *Other traces*
Traces that failed to authenticate (i.e., branch B3 not taken) or received invalid commands (i.e., sequence number not validated at B4 or unimplemented switch). Discarded as they do not contain any interesting Windows APIs.

For each trace we maintain three records: (a) a list of executed APIs and their call sites[1], (b) a concrete assignment for each symbolic buffer D_i manipulated by one or more API in the trace, and (c) a list of branches taken along the explored path. For instance, the first trace reports that the client instantiates a connection to a C2 server identified by the `sockaddr` structure `SERVER_ADDR`, performs two consecutive `read` operations, and executes the `WinExec` API. The trace provides concrete assignments for buffers D_1, D_2, and D_3, obtained by querying the solver before terminating the exploration. When buffers used as arguments for known APIs contain unconstrained or loosely constrained symbolic values, we add constraints to make the solver produce an answer meaningful for a human agent. For instance, we ask the solver whether symbolic buffer D_3 can hold a path to some executable. Taken branches are split across the two invocations of `read`. In our traces `read` yields symbolic buffers D_1 and D_2, while D_3 is a substring of D_2.

Observe that the two traces cannot directly be merged to form a longer one involving both commands: the sequence numbering scheme would drop replayed packets for commands issued in different communication sessions.

Trace Analysis. To compactly represent the generated traces we use an APTA representation inspired by prior protocol reversing research [9]. An APTA is a tree-shaped, incompletely specified DFA (deterministic finite state automaton) [5]: its root represents the common initial state for the explored traces, while each path in the tree denotes a communication session (in other words, a trace). Each path ends with a final state that marks the end of a trace: such a state does not necessarily correspond to the end of communications with the C2 server, but may be part of a session that we explored only partially under the given resource budget. In our setting, internal nodes of the APTA other than the root represent *receive* operations, i.e., APIs such as the `read` function that

[2] Omitted for brevity, we use them to distinguish different calls to the same API.

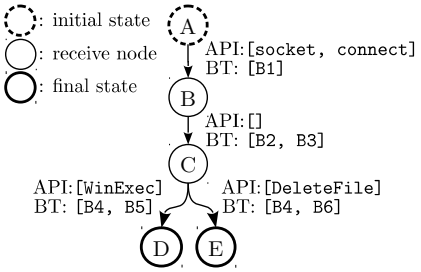

Fig. 3: APTA for the artificial RAT.

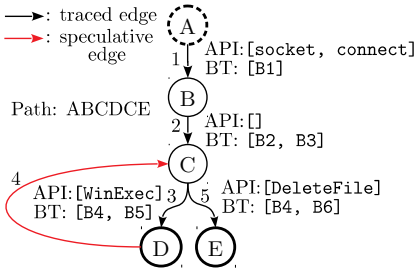

Fig. 4: Speculation step.

generate new symbolic inputs and thus identify incoming data buffers from the C2 server. Edges are characterized by two records extracted from traces: the interesting APIs invoked (API) and the branches taken (BT) between the two actions encoded by the two nodes of the APTA. Paths with the same prefix of nodes and edges get merged by the tree representation.

Previous research [9] uses inferred message types for edge labels, while we do not assume to know such information. Instead, we choose labels according to the relevant effects from receiving a message, that is, the interesting APIs executed in the trace. Our goal is clearly different than the one pursued in [9]: we are not interested in building a specification of the communication protocol, but rather in generating a synthetic C2 server for in-vivo RAT dissection.

Figure 3 shows the APTA built for our artificial RAT. Given the two traces generated in the initial step, we construct a tree characterized by two distinct paths with a common prefix. Node A is the initial state. Nodes B and C represent the two invocations of the `read` function executed in the body of the dispatcher loop across two loop iterations. Nodes D and E denote the final states for the first trace and the second trace, respectively. Edge AB is characterized by the invocation of `socket` and `connect` and by the taken branch B1. Edge BC reports only the taken branches `B2` and `B3` since no interesting API appeared in the traces between the two `read` invocations. Edges CD and CE represent what has been recorded in two traces from the last invocation of the `read` function (node C) and the end of each trace.

Speculation. We would like to identify a theoretical path that could drive symbolic execution into generating a single trace that exposes all the commands implemented by the RAT under analysis. As the list of such commands is not know a priori, we can try and learn from traces generated in the first step, leveraging the practical observation that several real-world RATs are implemented through an infinite dispatcher loop: although we expect a few commands to sink the communication (e.g., shutdown command), a significant fraction of them could be executable one after the other. Instead of using heuristics to identify commands, we limit our reasoning to the APIs observed in the initial step. In

other words, we characterize commands based on the set of APIs that could be executed by the RAT in response to some network data received from the C2 server. Hence, the aim of this step is to speculate over the feasibility of an execution path in the RAT able to execute the entire set of APIs observed across several traces during the first step of our approach.

We start by choosing the order by which we would like to visit interesting edges in the APTA. This order could be arbitrary or based on hints provided by an analyst[2]. To make exploration easier, we require that subsequent edges within alternative paths inside the APTA appear consecutively (i.e., edges are not interleaved). To build a path we start from the root node of the APTA and visit edges according to the chosen order until a leaf node is met. We then add a speculative edge from the leaf node to the source node of the next edge that we would like to visit. We continue to visit edges according to the order until another leaf is reached. We repeat the process of adding speculative edges as long as needed, until all the interesting edges are covered by the generated path.

For the APTA of Figure 3 we choose to visit edges in the following order: AB, BC, CD, CE. Edge AB has to come before BC and CD due to their ordering in the first trace. Similarly, BC comes before CD. The ordering between CD and CE is arbitrary. To build a valid path covering all edges, we start from the root node A and follow edges in order until the leaf D is met. To hit the remaining edge CE we add a speculative edge between nodes D and C, and eventually we visit E to cover CE. Hence, the trace that we would like to generate in the next phase should follow the path: ABCDCE. Figure 4 shows the path on the APTA after adding the speculative edge.

Validation. To validate the feasibility of the speculatively generated path we resort to symbolic execution. Differently from the first step, we can now exploit knowledge from past executions. Any non-speculative edge in the path is found in the APTA too, so it represents a portion of a trace recorded in the initial step: the taken branches associated with non-speculative edges can thus be used to guide symbolic execution when attempting to cover them.

We carry out the symbolic execution following three different exploration strategies: *strictly-branch-guided*, *target-guided*, and *loosely-branch-guided*.

The first strategy is only used at the beginning when starting the symbolic execution from the same point used in the first step. Since the speculative path begins with a prefix that *strictly* coincides with one of the traces generated during that step, the exploration is guided along the branches that were taken in each edge along that prefix. Assuming the execution is deterministic, the control flow should reach the first node in the speculative path that corresponds to a leaf in the APTA following exactly the same branches as in the original trace, i.e., the symbolic engine does not yield alternative execution paths.

[2] An analyst may desire to test whether the RAT can execute a speculated sequence of APIs that they build by combining insights from previous observations.

Upon reaching the first leaf, we switch to the *target-guided* exploration strategy switches. Since the execution should now aim at covering a speculative edge, there is no record of taken branches to reach it from past executions. However, the exploration has a well-defined target location: the node reached by the speculative edge. Hence, we now allow for a strategy where multiple alternative paths can be explored by the engine, and see whether within a given time and memory budget at least one such path reaches the target location. When one is found, the exploration continues along it and we switch to the third strategy.

In the *loosely-branch-guided* strategy we are trying to cover a non-speculative edge from the current execution path. Although this edge is present in the APTA and thus a list of taken branches is available, we cannot expect execution to strictly follow it: as the internal program state may have changed along the path, there is no guarantee that the RAT can now take branch decisions observed in the exact same manner as in the original trace behind that edge. We thus try to drive execution towards such list of branches using a *loose* enforcement: the strategy is designed to tolerate up to k divergences (i.e., different branch decisions) with respect to the original branches. If at least one execution path reaches the next leaf in the speculative path, we discard any other alternative paths and switch back to the *target-guided* strategy, aiming at covering the next speculative edge in the path. We repeat this process as long as needed to validate the entire path.

During this process the *target-guided* and *loosely-branch-guided* strategies may possibly fail. In this case, our technique goes back to the speculation phase, marks as invalid the speculative edge that led to the failure and tries a different speculation. In particular, for such an invalid edge (s, t) we replace it with a new edge between node s and the nearest ancestor of node t, i.e., its parent. We then recompute the speculative path using the new edge in place of the old one and we attempt validation. Observe that another failure would lead us to consider the second nearest ancestor for t and so on. In case of repeated failures when considering other of its ancestors, we update the speculative path by dropping any edge along the sub-path from t to the next leaf. After completing the validation phase, if any edge present in the APTA has been dropped in the process we go back to the speculation phase and compute an alternative additional path covering the dropped edges. As every edge in the APTA gets covered by at least one speculative path, a pathological exploration sequence may lead to a number of speculative paths equal to the number of traces obtained in the first step.

Given the speculative path ABCDCE for our artificial RAT, we start the symbolic exploration from node A using the strictly-branch-guided strategy. The execution path follows branches B1, B2, B3, B4 and B5 (just as expected as we are replaying the first recorded trace) and reaches node D, which is a leaf in the APTA. From now on, the exploration switches to the target-guided strategy in order to reach node C. Assuming a time budget sufficient to cover a few instructions, the symbolic engine can lead the execution path to successfully reach C. Then, the engine switches to the loosely-branch-

guided strategy, trying to drive the exploration towards branches B4 and B5. At least one symbolic execution path can reach node E (leaf) at the end of our speculative path, thus terminating the validation step. No APTA edge was dropped from the initial speculative path, thus no extra speculative paths are required. The trace generated for the speculative path is:

(a) API: `socket(...)`, `connect(..., SERVER_ADDR, ...)`, `read(..., D1, 64)`, `read(..., D2, 64)`, `WinExec(D3, 0)`, `read(..., D4, 64)`, `DeleteFile(D5)`
(b) buffers: `*D1 = "MAGIC_STR"`, `*D2 = "12Program.exe"`, `*D3 = "Program.exe"`, `*D4 = "22File.doc"`, `*D5 = "File.doc"`
(c) branch taken: `[B1], [B2, B3], [B4, B5], [B4, B6]`

C2 Server Generation. We synthesize a distinct C2 server for each trace generated from a speculative path. Each server sends back to the RAT the network buffers reported in the trace. To this end, we use network APIs that carry out symmetric functionalities for those that get executed in the RAT client.

Given the network APIs and the buffers listed in the trace generated during the previous step for our RAT, the C2 server implementation is characterized by the following flow of actions:

```
s1 = socket(...); bind(s1, ...); // symmetric to socket()
s2 = accept(s1, ...);            // symmetric to connect()
write(s2, D1, 64);               // symmetric to read(..., D1, ...)
write(s2, D2, 64);               // symmetric to read(..., D2, ...)
write(s2, D4, 64);               // symmetric to read(..., D4, ...)
```

The analyst sets up redirection for connections from the RAT to the server instance, then in-vivo analysis can take place in a debugger or other tools.

2.3 Discussion

A number of issues may hinder the applicability of our approach when considering real-world RATs. In the following we discuss relevant challenges we identified.

Encryption. Communications with C2 servers may be protected by crypto schemes. Although this may seem a showstopper at first—as SMT solvers used in symbolic execution cannot defeat robust encryption—we may still be able to analyze several real-world such instances. RATs often use symmetric crypto functions with keys embedded in the binary or sent by the server: since symbolic execution can handle both scenarios, our approach may sustain C2 server reconstruction for such RATs. Asymmetric crypto schemes are instead likely to make symbolic execution fail at reasoning over exchanged data. However, if the sample

is using standard crypto APIs we could extend our approach to perform function call interposition, rewriting API arguments to use custom private and public keys. For instance, a symbolic engine may model the Windows `CryptDecrypt` API internally in order to return a valid content for an encrypted network buffer. To make the synthesized server work, similar hooks should be applied also when running the RAT within the analyst's environment.

Nondeterminism and other input types. Another challenge is represented by nondeterministic factors in the execution. Recorded traces during phase one could not be directly *repeatable* [17] due to nondeterminism in OS interactions, the network or other external factors such as time sources. Since we implemented our approach on top of S2E [6], a framework for whole-system analysis, some of these issues could be mitigated in practice. However, the problem of limiting nondeterminism in the RAT execution when running it in the analyst's analysis environment remains an open problem.

A crucial assumption behind our approach is that the execution of interesting APIs causally depends on the receival of specific commands from the C2 server. This assumption may be not always be true and several factors can thus undermine our reasoning. In general, network may not be the only source of input: as we will see in Section 3, real-world RATs may use other data coming from the environment (e.g., obtained through a system call) as an input. Our approach builds on top of the assumption that any input source can be identified and marked as symbolic. In practice, this may require additional effort from the analyst. Also, our model should be extended to account for concurrent communications with the C2 server carried out by using, e.g., multiple threads.

Number of paths in the APTA. When constructing an APTA using the approach presented in Section 2.2, we may get an extremely large number of paths inside the APTA. Unlike our toy example, a command handler of a real-world RAT may embed a logic made of several conditional branches: unfortunately, each such branch can make the exploration fork, yielding several traces for the same command and increasing the number of paths in the APTA. To mitigate this problem, we merge APTA edges that refer to the same source and target receive nodes and report the same list of interesting APIs. In other words, we merge edges that differ only for the taken branches. To avoid losing useful information when merging, we define the `BT` record as a set of lists, where each list identifies the branches taken in some trace. Besides reducing the size of the APTA, this allows our approach to consider multiple—alternative—ways of guiding a path over an edge during the validation step, possibly increasing the chances of covering a command.

Speculation. Finally, we point out that the order by which during the speculation phase we pick edges of the APTA to be visited plays a crucial role. The interplay between commands may make a chosen speculative path unfeasible, yielding in the worst case to traces just as short as those we were able to record during the first step of our approach. Nonetheless, testing and validating any possible speculation decision for path construction is not feasible as we would be dealing with a combinatorial choice problem.

3 Experimental Evaluation

In the following we discuss a preliminary experimental evaluation of our approach considering two real-world RATs. We prototyped our technique in S2E [6] to perform symbolic execution on the entire software stack (including kernel and libraries), and extended it with more than 70 hooks for Windows APIs—used to inject symbolic data in case of network communications and to track interesting APIs—and the three exploration strategies discussed in Section 2.2.

We run the experiments on a server equipped with two Intel Xeon E5-2630 v3 CPUs (16 cores in total, @2.40GHz) and 256GB of RAM, running Debian Linux 9. We use a budget of 6 hours and 32 GB of RAM for the trace generation step, and one of 12 hours and 64 GB of RAM for the validation step.

3.1 NetWire

NetWire is a RAT with keylogging capabilities sold on the black market. It exists in several variants and has been used by criminals since 2012. In 2016 this RAT received particular attention from security vendors after taking part in a large campaign for stealing payment card data [16]. The variant we considered [15] allows an attacker to perform 51 commands on the infected machine, e.g., sending a file, stealing browser credentials, or taking a screenshot of the victim's desktop. Figure 5 (Appendix A) provides an annotated CFG for its dispatcher.

Trace generation. Starting the exploration from the executable's entry point would likely lead to the generation of a large number of paths in the symbolic executor, exhausting the resource budget well before reaching the dispatcher loop. For this reason, as in [2] we assume that the analyst can identify the thread executing the loop and provide hints for steering the exploration toward this component. Since in NetWire it is implemented as a `switch` statement with a large number of cases, detecting it in an initial inspection was straightforward. Our prototype generated over 2000 traces, covering 41 commands while 10 led to internal crashes. An excerpt[3] from a valid trace is the following:

(a) API: ..., `socket(...)`, `connect(...)`, ..., `time(T)`, ..., `send(...,`D_1, ...), ..., `recv(..., `D_2`, ...)`, `recv(..., `D_3`, ...)`, `fopen(`D_4, ...), `send(...,`D_5, ...), ...

(b) buffers: *D_1= `f(T)`, *D_2= `SEED_IV`, *D_3= `{0x3e, ...}`, *D_4= `"C:\Users\...\Opera\profile\wand.dat"`, *D_5= `{...}`

When analyzing the entire trace, we can learn several interesting facts about NetWire. First, it uses a raw TCP socket for communicating with the C2 server. Second, it sends to the server a buffer that contains some data `f(T)` derived from a timestamp `T` obtained using the function `time`. The server sends back a buffer that likely contains a seed and an initialization vector, internally used by NetWire in combination with a statically embedded password to generate a

[3] Addresses of taken branches seem of little interest and are thus omitted.

symmetric key. This key is then used during communications to encrypt data using a custom implementation of AES. As discussed in Section 2.3, symmetric encryption can be handled by our approach when the key is embedded in the binary or is dynamically generated with a computation. However, to make the trace *replayable* outside our environment the `time` function should be hooked in order to produce the same value used in the symbolic exploration. Another interesting element from this trace is that NetWire implements a command (`0x3e`) that sends to the server the content of profile files for the Opera browser.

Trace analysis, speculation, and validation. The APTA built from analyzing the traces is depicted in Figure 6 (Appendix A). To make the representation compact, we merged paths related to the same command (see Section 2.3) and discarded traces where no interesting APIs were observed. The resulting APTA contains 29 distinct paths, covering 30 commands (since command `0x4` is a prefix for all the paths). We discarded traces related to 11 commands for which no interesting APIs were recorded. Figure 6 shows the path chosen in the speculation step: the path was successfully validated during the fourth step, generating a single execution trace able to cover all 30 commands.

C2 server generation. To synthesize a C2 server instance for NetWire, our prototype programmatically generated a Python program template using wrappers for network APIs such as `socket`, `bind`, `accept`, `read`, and `write`. When generating the server, our prototype reports that the analyst should hook (e.g., using debugger scripting or by rewriting API results in a sandbox) the invocation of the `time` function that takes place at a specific call site inside the sample.

3.2 GoldSun

The first specimen of the GoldSun RAT dates back to 2004: over the years, the RAT has infected machines in over 60 countries [22]. We considered a variant [2] that injects code in Windows Explorer to remain stealthy and lets the attacker perform 16 types of commands on the victim's machine, such as executing a file or stealing one. Figure 7 (Appendix B) provides an annotated version of the control flow graph (CFG) for the dispatcher loop of the variant we analyzed.

Trace generation. Similarly as for NetWire, we rely on the initial guidance of the analyst to reach the dispatcher loop: our prototype could generate more than 300 traces originating in it, exposing all the 16 commands implemented by the RAT. An excerpt from one of the traces is the following:

(a) API: ..., `InternetOpenA(...)`, `InternetConnectA(..., D1, 80,...)`, ..., `InternetReadFile(..., D2, 4096, ...)`, ..., `WinExec(D3, 0)`, ...
(b) buffers: `*D1= "mse.vmnat.com"`, `*D2= XOR(0x45, "@@5File.exe")`, `*D3= "File.exe"`

where $*D_1$ is the server domain used by the C2 server (listening on port 80) and $*D_2$ is the buffer for requesting the RAT to execute the program `File.exe` on the victim's machine using `WinExec`. Notice that $*D_2$ is encrypted with a single-byte XOR scheme (key 0x45) while buffer $*D_3$ is just a substring of decrypted $*D_2$.

Although such encryption scheme is rather weak, it is sufficient to hinder reuse of messages across different communication sessions: the C2 server can instruct the sample to change the key, making messages from one session possibly invalid for another. In addition to message format and encryption factors, classic network simulators like FireEye FakeNet-NG would be defeated by the authentication scheme using by this RAT, which requires the C2 server to execute the first command two times before allowing the execution of any other command.

Trace analysis, speculation, and validation. The APTA built after analyzing the recorded traces is depicted in Figure 8 (Appendix B). As for NetWire, we merged paths for the same commands and discarded uninteresting traces, yielding a tree with 15 paths (as the first command appears twice in each path).

To build the speculative path, we added to the approach presented in Section 2.2 the requirement for the path to cover the command 0x10 after the last speculative edge. This choice results from the observation that this command makes the dispatcher loop terminate, thus acting as a shutdown action. This reasoning could be easily integrated into the approach by devising a heuristic for detecting traces that terminate the execution, e.g., by using `exit`-like APIs.

Interestingly, we experienced several edge invalidations when performing the speculation step. While our original speculative path had all the speculative edges reaching the deepest receive node in the APTA (node K in Figure 8), the final validated path shows several speculative edges reaching other receive nodes (i.e., ancestors of K). This resulted from multiple failures during the symbolic exploration. Nonetheless, no interesting edge was dropped during the validation phase, resulting in a single speculative path (reported in Figure 8) able to execute all the 16 commands implemented by GoldSun.

C2 server generation. The sample communicates with the C2 server using the HTTP protocol (see, e.g., the `InternetConnectA` API in the trace excerpt above). Our prototype can synthesize C2 server instances for this protocol in the form of Python Flask applications. For GoldSun, the C2 server is mainly implemented by handling request on the route `/httpdocs/mm/ <victim_id>/Cmwhite`, where `victim_id` is a unique identifier for the victim computed in the sample by concatenating the hostname and the MAC address of the victim's machine. When generating the server, our prototype reports that the analyst should set the host name and the MAC address in the analysis environment to same values used in the symbolic exploration to prevent path divergences in the execution.

4 Related Work

Symbolic execution and protocol reverse engineering techniques constitute the backbone of our approach. We briefly review works targeting such research.

Symbolic execution. Symbolic execution is a program analysis technique pioneered in the '70s. The technique is used to explore multiple paths of a program by using symbolic instead of concrete inputs and it is widely used by the cybersecurity community [20, 21, 19, 18]. Due to space limitation, we refer the reader to the work by *Baldoni et al.* [3] for a complete treatment of the subject.

Protocol reverse engineering. In the last two decades, protocol reverse engineering has received large attention from the research community. A recent survey [12] divides works on this topic in two fields:

Message format inference. The goal is to classify the messages exchanged between client and server and, for each identified class, to deduce the fields which compose a message from the class and the relations (if any) among these fields. The ultimate goal is to reconstruct the grammar that defines the structure of messages. Automatic approaches can be further categorized based on the type of inference applied. Techniques based on *network inference* analyze the packets exchanged between client and server. For instance, [4] uses sequence alignment algorithms to find similarities among the exchanged messages and exploits this knowledge to identify recurring patterns and field boundaries. [11] assumes that fields delimiters are known and uses a set of heuristics to classify the messages and to identify hierarchical relations between fields. Techniques based on *application inference* monitor not only packets, but also the code. [14] leverages the idea that an application manipulates logically-related fields in code portions that are close to each other. [9] uses instead a combination of system-call monitoring and taint analysis to deduce the field boundaries and possibly the semantics of the fields (e.g., if a field encodes a length).

Protocol grammar inference. The aim of such works is to reconstruct valid sequences of messages accepted by client and server. Message format inference is typically a prerequisite for this phase. *Active* inference approaches start from an inaccurate model and iteratively refine it by probing an application that implements the protocol; automaton learning algorithms are typically involved. For instance, [7] infers the protocol state machine of the MegaD C2 server by probing an active server using the L* algorithm [1]. [8] performs active protocol grammar inference in combination with concolic execution. *Passive* approaches deduce instead the protocol grammar by examining network traces. [9] constructs an automaton that models the protocol by processing sequences of captured messages that have been previously classified.

5 Conclusion

In this paper we have proposed new ideas for synthesizing a server counterpart for a RAT malware when only the client is available for inspection. As directions for future work, we plan to extend our automata representation to account for concurrent communications, and explore the limits of our implementation when dealing with more complex protocols with respect to packet format and interactions with the OS other than network-related ones.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (Nov 1987). https://doi.org/10.1016/0890-5401(87)90052-6, http://dx.doi.org/10.1016/0890-5401(87)90052-6
2. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C.: Assisting Malware Analysis with Symbolic Execution: A Case Study. In: Dolev, S., Lodha, S. (eds.) Cyber Security Cryptography and Machine Learning. pp. 171–188. CSCML '17, Springer International Publishing, Cham (2017)
3. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A Survey of Symbolic Execution Techniques. ACM Computer Surveys **51**(3), 50:1–50:39 (5 2018). https://doi.org/10.1145/3182657, http://doi.acm.org/10.1145/3182657
4. Beddoe, M.A.: Network protocol analysis using bioinformatics algorithms. Toorcon (2004)
5. Bugalho, M., Oliveira, A.L.: Inference of regular languages using state merging algorithms with search. Pattern Recogn. **38**(9), 1457–1467 (Sep 2005). https://doi.org/10.1016/j.patcog.2004.03.027, http://dx.doi.org/10.1016/j.patcog.2004.03.027
6. Chipounov, V., Kuznetsov, V., Candea, G.: The S2E platform: Design, implementation, and applications. ACM Trans. on Computer Systems (TOCS) **30**(1), 2:1–2:49 (2012). https://doi.org/10.1145/2110356.2110358
7. Cho, C.Y., Babi ć, D., Shin, E.C.R., Song, D.: Inference and analysis of formal models of botnet command and control protocols. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. pp. 426–439. CCS '10, ACM, New York, NY, USA (2010). https://doi.org/10.1145/1866307.1866355, http://doi.acm.org/10.1145/1866307.1866355
8. Cho, C.Y., Babić, D., Poosankam, P., Chen, K.Z., Wu, E.X., Song, D.: Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In: Proceedings of the 20th USENIX Conference on Security. pp. 10–10. SEC'11, USENIX Association, Berkeley, CA, USA (2011), http://dl.acm.org/citation.cfm?id=2028067.2028077
9. Comparetti, P.M., Wondracek, G., Kruegel, C., Kirda, E.: Prospex: Protocol specification extraction. In: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy. pp. 110–125. SP '09, IEEE Computer Society, Washington, DC, USA (2009). https://doi.org/10.1109/SP.2009.14, https://doi.org/10.1109/SP.2009.14
10. Coppa, E., D'Elia, D.C., Demetrescu, C.: Rethinking Pointer Reasoning in Symbolic Execution. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering. pp. 613–618. ASE '17, IEEE Press, Piscataway, NJ, USA (2017). https://doi.org/10.1109/ASE.2017.8115671, http://dl.acm.org/citation.cfm?id=3155562.3155638
11. Cui, W., Kannan, J., Wang, H.J.: Discoverer: Automatic protocol reverse engineering from network traces. In: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium. pp. 14:1–14:14. SS'07, USENIX Association, Berkeley, CA, USA (2007), http://dl.acm.org/citation.cfm?id=1362903.1362917
12. Duchêne, J., Le Guernic, C., Alata, E., Nicomette, V., Kaaniche, M.: State of the art of network protocol reverse engineering tools. Journal of Computer Virology and Hacking Techniques **14** (01 2017). https://doi.org/10.1007/s11416-016-0289-8
13. Jiang, D., Omote, K.: An approach to detect remote access trojan in the early stage of communication. In: 2015 IEEE 29th International Conference on Advanced Information Networking and Applications. pp. 706–713 (March 2015). https://doi.org/10.1109/AINA.2015.257

14. Lin, Z., Jiang, X., Xu, D., Zhang, X.: Automatic protocol format reverse engineering through context-aware monitored execution. In: 15th Symposium on Network And Distributed System Sexurity (NDSS) (2008)
15. Luxembourg, C.I.R.C.: TR-23 Analysis - NetWiredRC malware (2014), uRL https://www.circl.lu/pub/tr-23/
16. SecureWorks: NetWire RAT Steals Payment Card Data (2016), uRL https://www.secureworks.com/blog/netwire-rat-steals-payment-card-data
17. Severi, G., Leek, T., Dolan-Gavitt, B.: Malrec: Compact full-trace malware recording for retrospective deep analysis. In: Giuffrida, C., Bardin, S., Blanc, G. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 3–23. Springer International Publishing, Cham (2018)
18. Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G.: Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware (01 2015). https://doi.org/10.14722/ndss.2015.23294
19. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: Sok: (state of) the art of war: Offensive techniques in binary analysis. pp. 138–157 (05 2016). https://doi.org/10.1109/SP.2016.17
20. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: Bitblaze: A new approach to computer security via binary analysis. In: Proceedings of the 4th International Conference on Information Systems Security. pp. 1–25. ICISS '08, Springer-Verlag, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89862-7
21. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution (01 2016). https://doi.org/10.14722/ndss.2016.23368
22. Villeneuve, N., Sancho, D.: The "Lurid" Downloader. Trend Micro Incorporated (2011), uRL: http://la.trendmicro.com/media/misc/lurid-downloader-enfal-report-en.pdf (retrieved March 2017)

A NetWire RAT

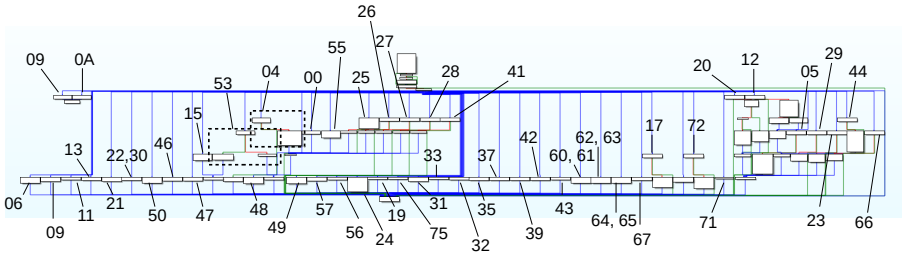

Fig. 5: NetWire – Control flow graph of the RAT's dispatcher loop.

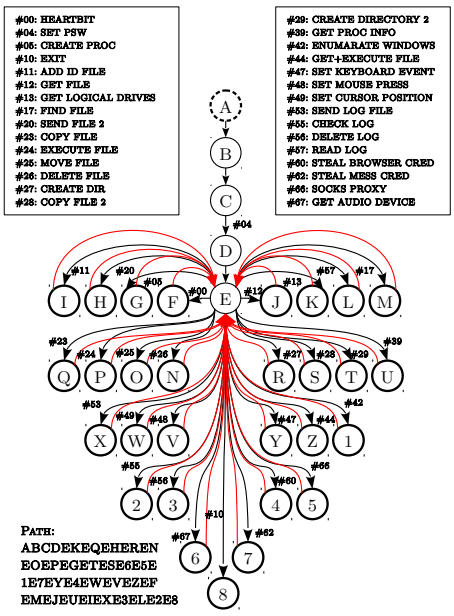

Fig. 6: NetWire – APTA with speculative edges after completing the validation step.

B GoldSun RAT

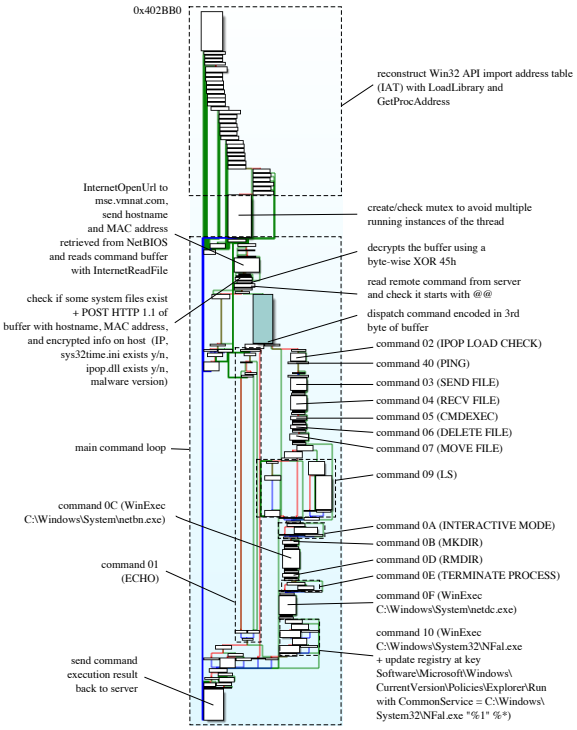

Fig. 7: GoldSun – Control flow graph of the RAT's command processing thread [2].

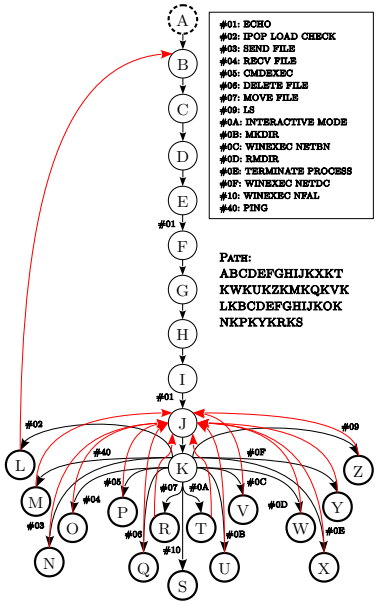

Fig. 8: GoldSun – APTA with speculative edges after completing the validation step.